NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN ALGEBRAIC SPECIFICATION LANGUAGE
AND A SYNTAX DIRECTED EDITOR

by

Norvell L. Lilly
December 1984

Thesis Advisor: D. L. Davis

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

An Algebraic Specification Language and a Syntax Directed Editor

5. TYPE OF REPORT & PERIOD COVERED

Master's Thesis
December 1984

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

Norvell L. Lilly

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Naval Postgraduate School
Monterey, California 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Naval Postgraduate School
Monterey, California 93943

12. REPORT DATE

December 1984

13. NUMBER OF PAGES

74

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Syntax-directed editor, algebraic specification
formal specifications

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The rising cost of software has created a demand for methodologies which will allow the creation of portable software. Formal specification methods have been used to increase the portability of software, permit a degree of verification and validation, and lessen the burden of maintenance. The underlying theory for most specification methods, however, make them difficult to use and frequently are not applicable to many problems. (Continued)

ABSTRACT (Continued)

Formal specifications based on initial algebras provide a framework for precisely defining program behavior and avoid the problems of informal specification methods. This thesis presents a specification language based on initial algebras, describes the various parts of specification produced by the language and describes an experimental syntax directed editor which uses the language's grammar.

Approved for public release; distribution is unlimited.

An Algebraic Specification Language and a Syntax Directed Editor

by

Norvell L. Lilly
Lieutenant Commander, United States Navy
B.S., University of Pittsburgh, 1973

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1984

ABSTRACT

The rising cost of software has created a demand for methodologies which will allow the creation of portable software. Formal specification methods have been used to increase the portability of software, permit a degree of verification and validation, and lessen the burden of maintenance. The underlying theory for most specification methods, however, make them difficult to use and frequently are not applicable to many problems. Formal specifications based on initial algebras provide a framework for precisely defining program behavior and avoid the problems of informal specification methods. This thesis presents a specification language based on initial algebras, describes the various parts of specification produced by the language and describes an experimental syntax directed editor which uses the language's grammar.

TABLE OF CONTENTS

I. INTRODUCTION 9

II. SPECIFICATIONS BASED ON ALGEBRAS 12

A. A REVIEW OF ALGEBRAS 12

B. SIGMA OR MANY SORTED ALGEBRAS 14

C. BOOLEAN AS A SIGMA ALGEBRA 16

III. A SPECIFICATION LANGUAGE 21

A. SPECLANG GRAMMAR 22

B. PARTS OF A SPECIFICATION 25

1. Header 26

2. Signature Part 26

3. Axiom Part 32

4. Speclang Modifiers 33

5. Parameterized Modules 37

IV. SPECIFICATION EDITOR 41

A. AN EDITING SESSION 42

1. "s" Selection 43

2. "i" selection 45

3. "b" and "f" Selections 45

4. "u" and "d" Selections 46

5. "m" Selection 46

6. "e" Selection 46

7. "r" Selection 47

B. DESIGN OF SPECED 48

1. Early Design Decisions 48

2. Initialization 49

3. Main Body Loop 56

4. Termination 66

V. SUMMARY 67

APPENDIX A: SPECLANG GRAMMAR 68

LIST OF REFERENCES 73

INITIAL DISTRIBUTION LIST 74

LIST OF TABLES

I. Node Types 51

II. Routines called by Main Body Loop 57

LIST OF FIGURES

2.1 Mapping of Integers 13
2.2 A Constant Mapping 14
2.3 The Push Operator 15
4.1 Display of the Rule <module_spec> 43
4.2 Selection of a Non-terminal 44
4.3 A Production Rule Tree 50
4.4 Selection of <spec_id> 59
4.5 Selection of the Option <param_block> 61
4.6 Selection of an Alternation Expression 63
4.7 Reselecting the <param_block> Option 65

I. INTRODUCTION

Software portability, as defined by Poole and Waite [Ref. 1], is the measure of the ease which a program can be transferred from one environment to another; if the effort required to move the program is much less than that required to implement it initially, and the effort is small in an absolute sense, then that program is highly portable.

Software portability has become an area of intense research as the "software crisis" continues to gain momentum. The less portable a program is, the more it will cost in terms of time and money to transfer it to another machine. One important impact of the lack of portability is that there will be little incentive to create truly user friendly environments since the cost of producing such a system can not be amortized over many machine implementations.

There have been many attempts to solve the portability problem. Many approaches have failed, some approaches have achieved limited success, few have provided a broadbased methodology for achieving portability. High level languages were one of the first attempts at resolving the problem. They, however, were either too small to be useful, and consequentely extended, or too large and therefore subseted. Even if the language achieved a great deal of consistency over many implementation, programs were still designed with non-standard, machine dependent features.

Decompilers and language translators were developed but their complexity and poor reliability discouraged further development

The use of specifications to define the behavior of a program is a method which offers a way to solve the portability problem. Although more work is required to code from a specification, the description of program behavior can be more precise and implementation independence is possible. Complications arise, however, when the specification language is ambiguous. Formal specification languages based on mathematical principles alleviate this problem but often are more difficult to use and frequently larger than the programs they specify. Despite the difficulty in their use, formal specifications offer the greatest degree of freedom from implementation without ambiguity.

A specification methodology based on initial algebras promises to be a viable answer to the portability problem without the drawbacks of most formal specification techniques.

Algebras have a wide range of applications. Many researchers have proposed treating abstract data types as algebras [Ref. 2], [Ref. 3], [Ref. 4]. Algebras are particularly well suited for describing data types; data types consist of data elements and operations on the data elements which is essentially the definition of an algebra. Fasel, in his thesis [Ref. 5], noted that if algebras can be used to specify data types, then the next step would be to use them to specify languages; a program is composed of instructions which can be expressed using algebras. Furthermore, research at the Naval Postgraduate School is aimed at using algebras to specify an abstract machine. A machine can be described using algebras since the execution of instructions causes the machine to change state. Algebras are used to define the effect each instruction has on the state of the machine.

Put simply, by using algebras, formal specifications can be produced which are truly independent of an implementation; many implementations of the specification can be created which emulate a formal algebraic specification. In addition, it is possible and more reasonable to prove the correctness of an implementation developed from an algebraic specification.

If the research at the Naval Postgraduate School is successful in specifying an abstract machine, software can then be targeted for the abstract machine and freed from machine dependence. A simpler layer of software will translate the abstract machine commands into a particular machine code. A tremendous savings can be realized when complex software, such as software that supports a user friendly environment, is targeted for this machine. Futhermore, it may be feasible to determine if one abstract machine is equivalent to another because of the precise algebraic specifications. Equivalent abstract machines would imply a way of translating software to different machines.

The intent of this thesis is to define a suitable language for an algebraic specification which minimizes the problem with using this methodology, and to implement an experimental syntax directed editor using this language. Chapter two provides the background theory of specifications based on algebras. Chapter three describes the various parts of an algebraic specification and the corresponding parts in the proposed language. Chapter four describes a syntax directed editor for the language.

II. SPECIFICATIONS BASED ON ALGEBRAS

The purpose of this chapter is to provide an introduction to algebras used for specifications. Goguen et al's work using algebras to specify abstract data types provides the basis for the language presented in the next chapter [Ref. 2]. A review of definitions and terms used in the algebras they discuss is worthwhile before introducing the specification language.

A. A REVIEW OF ALGEBRAS

An algebra consists of a set, called the carrier of the algebra, an operator defined on the carrier and distinguished elements of the carrier, called the constants. They are normally represented using a bracketed tuple. For example, the addition operator on integers would be expressed as follows:

$$<I,+,0>$$

The carrier set can be of any one type we wish to manipulate such as real numbers, integers or character strings. For the addition operator on integers, the carrier set is identified by the capital letter I and the elements of the carrier set are the integers. The distinguished element of I is zero.

An operation is a mapping taking one or more elements in the carrier to an element in the carrier. For example, the addition operator on integers will take two integers and map them to an integer as shown in Figure 2.1 .

It is common practice to refer to a specific class of algebras. The members of a particular class have the same signature or "arity" in their operators [Ref. 2]. For instance, the subtraction, addition and multiplication operators on integers are members of the same class since they have the same signature; they map two integers to one integer.

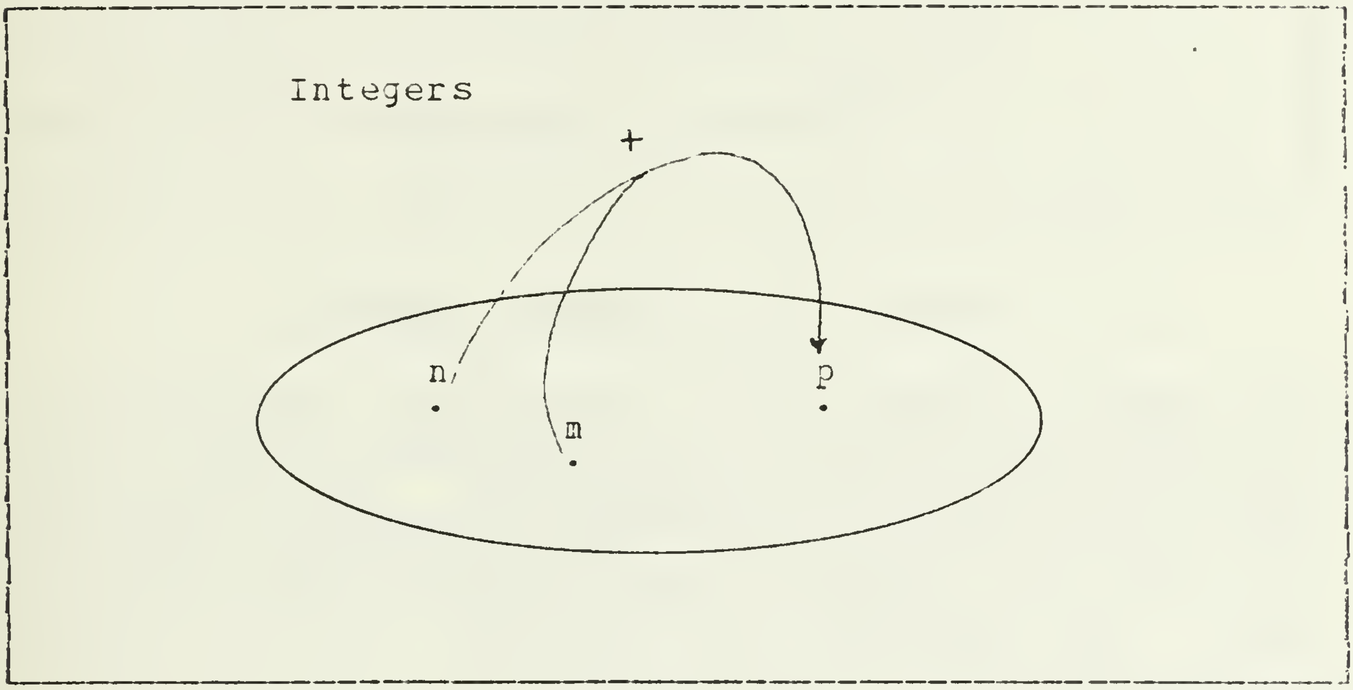


Figure 2.1 Mapping of Integers.

The constants or distinguished elements of the carrier usually have special properties. The number zero in the integers has a special property in regards to the addition operator; any number added to zero will map back to that number as shown in Figure 2.2 . Algebraic specifications will not explicitly deal with distinguished elements. Instead, constant operators are used to describe the distinguished elements. This is done to achieve implementation independence in the specification.

An algebra is not usually interesting unless it has some specific structure. The structure is defined by axioms. Using the above example, the addition operator as defined

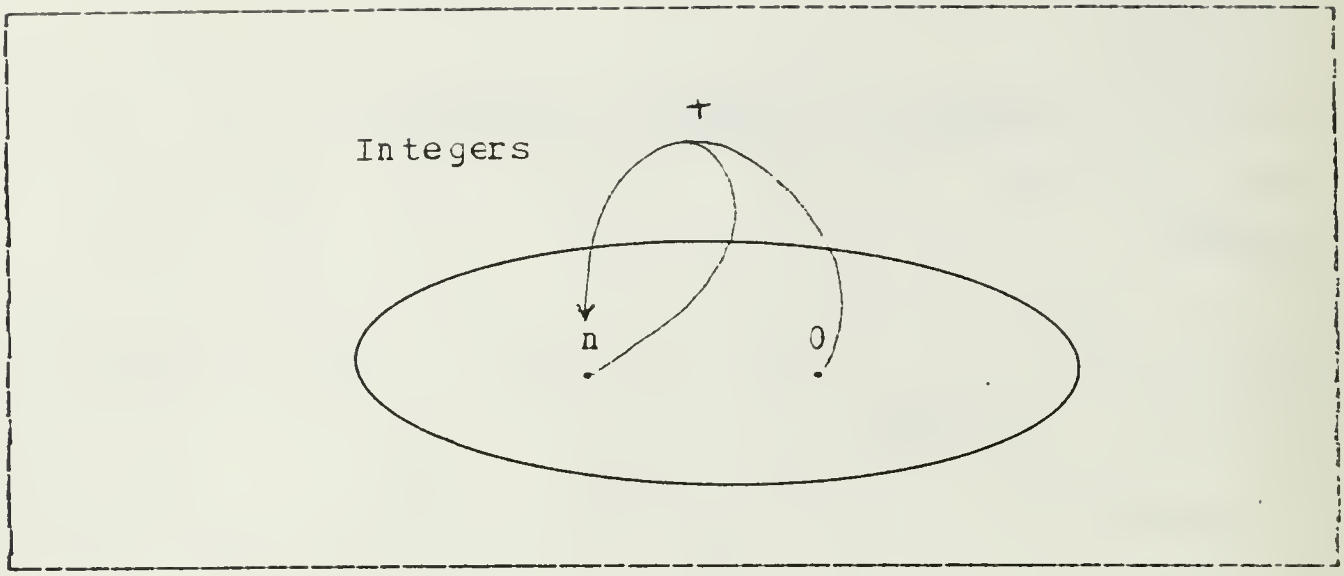


Figure 2.2 A Constant Mapping.

will not be useful if every application of the operator mapped an element to the same element. Therefore restrictions on the behavior of the addition operator are defined which limit the mapping. Later it will be shown how axioms define the behavior of operators in an algebraic specification.

B. SIGMA OR MANY SORTED ALGEBRAS

An algebra with one carrier set and mapping is not particularly useful for the purpose of specifications. For instance, in defining a stack as an abstract data type using algebras, we would not want to be limited to one type of data the stack contains such as a stack of integers. Instead we would like to talk about a stack of integers, reals, booleans, etc. Therefore in specifying an algebra for a stack, a 'sort' set would be defined. Each element of the sort set would be an index to a carrier set; the sort would identify the carrier set. A stack data type can then be defined as having a sort set as follows:

S={integers,boolean,stack,reals}

Each element in the set S represents a carrier. Now it is possible to define a stack of reals, integers, and boolean values. Notice that a stack also is a sort; a stack is a different object after operations are performed on it. As a result, stack is also a carrier set.

In addition to the many sorts of an algebra, it is desirable to have various operators on the sorts. Operators on the data type are defined by mappings from zero or more elements from carrier(s) to an element in a carrier set. Using the stack example, the push operator can be defined as:

push: integer,stack -> stack

The meaning of this operator is that given an element from each of the carriers identified by integer and stack, the push operator produces an element in the carrier identified by stack. This is depicted in Figure 2.3 .

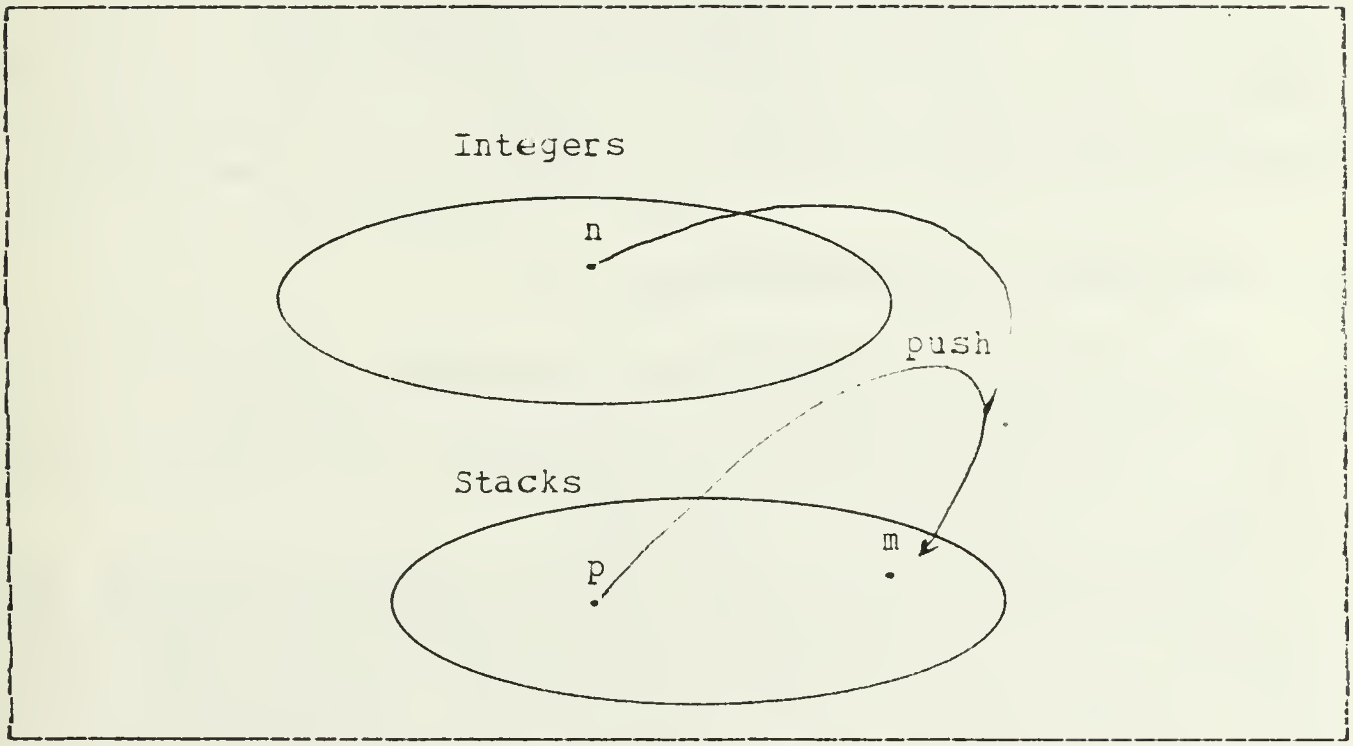


Figure 2.3 The Push Operator.

The operators in an algebraic specification are given by a sorted signature, Sigma, or operator domain, which is a family of sets. Each set of the family contains a particular domain for a group of operators. These all have the "arity" discussed in the previous section. Using the stack example, the arity Sigma(integer stack,stack) defines a domain for those operators which map an integer and a stack to a stack. The push operator would be a member of this domain. Any other operator which took an integer and a stack and mapped it to a stack would have the same arity.

The pop operator would be in the set of operators given by the arity Sigma(stack,stack). If there was an operator which removed two or more elements from a stack, it would also have the same arity. An interesting point here is that the integer which is removed is a side effect which must be described in the equations for the operator. All of the operators defined using the sort set are collectively known as the Sigma signature

Given the above, a specification algebra consists of a sort set S, a Sigma signature on the sort set, and named operators for each signature. Algebras defined as such are called many sorted algebras or Sigma algebras [Ref. 2].

C. BOOLEAN AS A SIGMA ALGEBRA

In order to explain the use of axioms, a development of the boolean data type as an algebra is presented. It will specify how an implementation of boolean logic should behave if used in a program or a machine.

An appropriate sort name for the boolean data type would be bool and the corresponding sort set S would be {bool}. Bool is the only sort type required to specify the boolean data type. The carrier set that bool identifies will contain two elements {T,F}. Although two specific elements of the carrier have been described here, an implementor may choose different representations for the elements of the carrier; a carrier with elements {0,1} would work equally well. In an algebraic specification, the elements of the carrier are not specifically defined so the implementor may choose any representation for carrier elements. In this way, a specifier can achieve implementation independence. However, for the purpose of clarity, the elements of the carrier will be used in the boolean example to define the operators.

The five boolean operators used in the specification will be as follows.

1. true - Returns a value of T.
2. false - Returns a value of F.
3. not - Returns the negaticn of a value.
4. implies - Implication
5. and - The logical 'and' cperator.

The following notation will be adopted for operators to describe the operand sorts and the resulting sort.

```
For Sigma(lambda,bool)
        true : -> bool
        false : -> bool

For Sigma(bool,bool)
        not: bool -> bool

For Sigma(bool bool,bool)
        and: bool,bool -> bool
        implies: bool,bool -> bool
```

The notation here is the same as that used to describe the push operator in the last section and will be the same notation used in the specification language presented later. For the signature Sigma(lambda, bool), the notation means the "true" and "false" operators produce an element in the carrier set identified by bcol. Lambda is used in conjunction with constant operators; constant operators always produce the same element and they appear to produce something from nothing. The "not" operator takes an element from the carrier set identified by bool and produces an element in the bool carrier set. The "and" and "implies" signature would be Sigma(bool bool,bool). That means, given two elements in the carrier set identified by bool, the "and" and "implies" operators produce an element in the carrier identified by bool.

The sigma algebra for boolean at this point is not very interesting. This is because the operators have no restrictions on their behavior. Consequently most implementations of boolean using just this specification would not be useful. For example, the "not" operator would be correct if it took any element in the carrier set and produced the same element. A more useful Sigma algebra would put restrictions on the operators. This is accomplished by introducing axioms or equations on the operators. In the Boolean sigma algebra, we would introduce the following axioms:

1. false = not(true)
2. not(not(v)) = v
3. and(true,v) = v
4. and(false,v) = false
5. implies(v1,v2) = not(and(v1,not(v2)))

Notice that the axioms were written without using a explicit reference to carrier set elements; by not using T or F the data type achieves implementation independence.

Implementations of the boolean data type are now required to mimick the axioms when operators are applied. For instance, the negation operator, "not", now behaves as expected and any implementation would have to reflect axioms one and two. Axiom one shows the effect "not" has on an element in the carrier set. Axiom two further defines the behavior of "not" by showing that any element in the carrier which is operated on by the "not" operator twice will produce the same element.

The "and" operator behaves as depicted by equations three and four; given any value v1 "anded" with the value produced from the "true" operator will result in that value. Implies is defined by axiom five as being the same as a combination of "not" and "and" operators using the same carrier values.

Note that with the exception of axiom one, all the axioms involve variables v, v1 and v2 which can be assigned all values from the carrier set defined for the operators "and" "not" and "implies". The axioms form the basis of all the expressions that can be created using the variables and members of the carrier set. This means that an expression which is built from other expression is permissable if it reduces to an element of the carrier set through the application of the axioms.

To illustrate this, suppose we had an expression

not(and(not(v1),v2))

If v1 is assigned the element T and v2 is assigned the element F, then the expression becomes the following by axiom cne:

not(and(F,F))

This further reduces to not(F) by axiom four. Axiom one implies that this reduces to T, an element of the carrier.

All the permissible expressions that can be constructed from the operators and elements of the carrier are collectively called the <u>term algebra</u>. The algebra that represents all the expressions that can be made up of variables and operators is called a <u>free algebra</u> [Ref. 2].

It is now possible to define a specification as a "presentation". A presentation consists of a sort set, an operator domain Sigma, and axioms on the operators. A presentation will be the basis for the specification language, Speclang ,presented in the the next chapter. A specfication developed from Speclang is comprise of one or more presentations and each presentation is known as specification modules.

III. A SPECIFICATION LANGUAGE

Liskov and Zilles [Ref. 4] have developed criteria for evaluating a formal specification language. Their work focuses on the underlying mechanism used in formal specification. They compared approaches using algebras, finite state machines, mixed mathematical disciplines, etc. The criteria used in their comparisons are:

1. Formality - The language must be precise and rigorous.
2. Constructability - It should be easy to construct a specification from the language.
3. Comprehensible - Is the specification relatively easy to understand?
4. Minimality - The specification generated should define its meaning with a minimum number of statements. One flaw of formal specifications is that they frequently require more statements than the program they specify.
5. Applicability - Can the language be used for a wide range of specifications?
6. Extensibility - A minimal change in concept results in a similar small change in a specification.

The rigorous underlying mathematics for algebras satisfy the formality criteria and the axioms restrict the amount of information required to explain desired behavior, thus satisfying minimality. Also, it appears that any desired change in behavior requires a minimum of additional axioms or minor modifications to the existing ones. As for applicability, algebraic specifications are being used for abstract machines, data types, and programming languages, which indicates a wide range of applications.

Algebras, in Liskov and Zille's opinion, satisfied all of the above criteria except comprehensibilty and constructability. These deficiencies can be reduced to a manageable level. The following proposed language, Speclang, will minimize these problem areas. In this chapter, it will be shown that a complex specification can be modularized; broken into smaller units which enhance constructability and comprehensibility.

There are many underlying issues involved in using algebras for specifications which have not been resolved. Issues such as proving specifications correct, infinite versus finite specifications, implementation and other complex issues will not be addressed. The following sections are to familiarize the reader with the grammar for Speclang, parts of a specification produced from the Speclang grammar and a brief explanation of the uses of the various parts.

A. SPECLANG GRAMMAR

Before presenting the various parts of a specification constructed from Speclang, an introduction to the grammar is presented. Appendix A contains the complete grammar for Speclang. The production rules are written in a modified Backus-Naur (BNF) notation. The meta-symbols in the production rules are used to form the construction of a specification and do not appear in the final specification. An explanation of the meta-symbols used to produce terminal strings in the grammar are as follows:

< > - A name enclosed by pointed brackets indicates a non-terminal in the grammar.

' ' - Strings in quotes indicate terminal strings.

( ) - Rounded brackets indicate the scope of a modifier symbol.

* - The Kleene closure modifier may follow either rounded brackets, terminals or non-terminals. It means that zero or more of the expressions, terminals or non-terminals may be produced.

+ - The alternation modifier is used like the Kleene modifier. It means that one or more of the expressions, terminals or non-terminals may be produced.

| - The selection modifier indicates a choice of non-terminals, terminals or expressions. It is used between two choices. In order to clarify the selection, rounded brackets enclose the selection.

? - The option modifier indicates that the terminal, non-terminal, or expression is optional and can be excluded.

-> - The production symbol indicates what the non-terminal on the left hand side can produce.

A production rule consists of a non-terminal followed by a production symbol followed by an expression. An expression is comprised of terminals, non-terminals, modifying symbols, and may include other expressions. The following production rule will be used as an example throughout the remainder of the thesis:

```
  <module_spec> -> <spec_header>
                   (<newline><indent><param_block>)?
                   <newline><indent><spec_body>
```

This rule defines a specification module. The module would begin with a specification header followed by an optional expression that contains a carriage return, indentation and a parameter block. Because it is an optional expression, it may be omitted. After the expression, another carriage return and indentation occurs followed by a specification body.

The grammar for Speclang includes indentation and carriage returns to permit a formatting of the specification. This was done to create a standard format which will, hopefully, increase the readability of a specification.

A specification is complete when all the strings in the specification are terminal strings. This is accomplished by starting with a nonterminal and substituting the expression appearing on the right hand side of its production rule. The remaining nonterminals are then substituted until all the strings are terminal strings. For example, starting with the nonterminal <module_spec> and substituting the right hand part of its rule results in the following:

```
     <spec_header> (<newline><indent><param_block>)?
             <newline><indent><spec_body>
```

The rule for a specification header is:

```
            <spec_header> -> 'SPEC' <spec_id> 'IS'
```

Its right hand side is substituted into the right hand side of the above strings to produce:

```
 'SPEC' <spec_id> 'IS' (<new_line> <indent> <param_block>)?
         <newline> <indent> <spec_body>
```

<newline> and <indent> can be interpreted as carriage return and a tab respectively, except when it followed by a modifier, and make the developing specification appear as:

```
   SPEC <spec_id> IS (<newline><indent><param_block>)?
        <spec_body>
```

In the lexical grammar for Speclang, <indent> is treated either as one or more tabs or spaces. This was done to permit the degree of indentation a specifier desires. Once a indentation has been selected, it should be continued throughout the specification. For instance, if one tab is used to indent a line, then every time indent is encountered one tab should be used.

Since <param_block> is opticnal it can be eliminated to produce:

SPEC <spec_id> IS
<spec_body>

The substitutions are continued until all strings are composed of terminal characters.

The first nonterminal string in Speclang is <comp_spec>. This denotes a complete specification. As mentioned in the previous section a specification developed from Speclang is made up of modules. Each module in itself is a specification. To reflect this the right hand side of the production rule for <comp_spec> is <module_spec>+. In other words, a complete specification is made up of one or more specification modules.

B. PARTS OF A SPECIFICATION

As mentioned in the previous section, a complete specification is made up of one or more specification modules. Each specification module is an algebraic specification, which can be viewed as a subspecification of the complete specification, and has three distinct parts - header, signature, and axiom parts. Each specification module is either "primitive", "extended" or "parameterized". Primitive modules do not import other specification modules while extended modules do import other specification modules to create a new specification. "Parameterized" modules are used to minimize a specification and can be either primitive or extended.

A specification is started with primitive modules. These modules are used to create other modules through the techniques described in the next sections. The last specification module defined in a complete specification, in effect, consists of all the other specification modules since it is built frcm the previous modules. The following section describes and discusses each part of a specification module.

1. Header

The header identifies the name of the specification and sometimes contains a "modifier". "Primitive" modules in Speclang contain no modifiers and the format is as follows:

```
SPEC <spec_id> IS
    {signature part}
    {axiom part}
```

<spec_id> is a slot for the particular name of that declared specification. The body of the specification, which contains the syntactic and semantic part, follows underneath the header. When used, the modifier is directly under the header. Its purpose is to import other specifications into the declared specification. The modifier is the primary method used to combat the complexity of an algebraic specification. It will be presented after the other parts of the specification are introduced.

2. Signature Part

The signature part of an algebraic specification contains the declarations for sorts and operators and defines the format and composition of the operators. It corresponds to the signatures discussed in chapter two.

a. Sort Declarations

In Speclang, the format for declaring sorts is:

```
SPEC <spec_id> IS
```

```
                    SORTS
                         <sort_id>;
                         <sort_id>;
```

<sort_id> is the slot for a particular sort name. As explained in the previous chapter, the sort will identify the carrier(s) used in the operators. All of the sorts declared under the header SORTS define the sort set.

b. Operators Declaraticn

The operators declaration can contain up to four different types of declarations:

1. Primitive
2. Derived
3. Hidden
4. Error

These operator type names are used as headers to each group of operatcrs declared in Speclang. The basic format for an operator is:

```
     <op_id> : <sort_id>, <sort_id> ... -> <sort_id>
```

An operation, as explained in chapter II, can map zero or more elements from a carrier set(s) identified by the sort name(s) to an element of a carrier set identified by the sort name.

A specification with sorts and operators would appear as follows:

```
     SPEC <identifier> IS
          SORT
               <sort_id>;
          OPERATIONS
               PRIMITIVE
                    <op_id>:-> <sort_id>;
                    <op_id>:<sort_id>,<sort_id> -> <sort_id>;
```

```
HIDDEN
    <op_id>: <sort_id> -> <sort_id>;
ERROR
    <op_id>: <sort_id> -> <sort_id>;
DERIVED
    <op_id>: <sort_id> -> <sort_id>
```

(1) Primitive Operators. Primitive operators are newly defined operators and cannot be constructed from any other operators such as the derived operators described next. Primitive operators may involve sorts from other specification modules and usually contain at least one sort from the specification in which they are declared.

(2) Derived Operators. Derived operators are operators which can be produced from other operators. They are declared so the implementor of the specification is aware that their existence is not essential. For instance, in the boolean specification, if the "not", "and" and "or" operators are defined, it is not necessary to include an "implies" operator since in predicate logic "implies" is equivalent to "not" A "and" B. The benefit in having a derived operator, such as "implies", is to minimize the specification and, in many cases, make the specification more comprehensible. In the boolean example, "implies" is a well understood operator and using it in other specification modules will make the modules more readable than if a combination of "not" and "or" were used. "implies" would be declared under the DERIVED header.

(3) Error Operators. The proper handling of errors is crucial to a specification. However, it is frequently the case that specifications for languages, data types, programs, etc. do not precisely define what should happen when an error is encountered. This is carte blanche for the implementor to do what he thinks is appropriate. In

the worst cases, nothing is done, permitting bizarre consequences.

A specification should address those circumstances where the operator may not make sense on the operands and define precisely how to handle them. For example, in the specification for a data type "stack" of integers, the sorts and operators would be as follows:

```
SPEC Stack IS
    SORTS
        int;
        stack;
    OPERATIONS
        PRIMITIVE
            new: stack;
            push: int, stack -> stack;
            pop: stack -> stack;
            top: stack -> int;
```

The "new" operator creates an empty stack and the "push" and "pop" operator behave as would be expected. The "top" operator allows the user to view the top element of the stack.

Problems occur when a user attempts to "pop" a "new" stack or view the first element of a "new" stack. These are valid operators since "new" returns a stack and the "pop" and "top" operators are defined on a stack.

It would seem reasonable to introduce a value "error" to each carrier set to define error conditions. Guttag [Ref. 3] proposed this approach. The result would be equations which specify what are error conditions. Axioms pop(new)= error and top(new) = error would seemingly resolve the problem. Guttag also required equations which defined the propagation of errors. Therefore push(error,s) = error and push(n,error) = error would be introduced into the specification.

Fasel has demonstrated the inadequacy of this approach [Ref. 5]. The valid equations top(push(n,error)) = n and top(push(n,error)) = error can be produced from the axioms which imply n = error for any n.

The approach adopted in Speclang is the one used by Fasel and developed by Goguen [Ref. 2]. This involves introducing error operators into the specification. These operators will produce error messages. In the stack operator, we would define error operators "topnew" and "underflow" which produce carrier values integer and stack respectively and also generate error messages. In Speclang, the error operator will be declared under the error operator title. Using the stack specification, the declaration would be as follows:

```
SPEC Stack IS
    SORT
        int;
        stack;
    OPERATIONS
        PRIMITIVE
            new: -> stack;
            push: int, stack -> stack;
            pop: stack -> stack;
            top: stack -> int;
        ERROR
            topnew: -> int;
            underflow: -> stack;
```

The axiom portion of the specification would use these operators to define error conditions.

(4) Hidden Operators. Unfortunately, it is not always possible to have just operators for the user. Some operators are required in order to define the user operators. Majster demonstrates this with a stack

specification which includes operators that allow the user to view any element of the stack without disturbing the stack. [Ref. 6]. In order to define these operators, it is necessary to define a "current position" which point to a particular element in the stack, not necessarily the top element. The new operators are "down" which moves the current position down one element in the stack, "return" which moves the current position to the top of the stack, and "read" which will give the value of the element at which the current position is pointing. Push, pop and top can only be done when the current position is on the top element. Using just the operators available to the user, it appears that an infinite number of equations are required to define the meaning of reading each element of the stack (if the stack is considered an infinite specification). The following are just a sample of the equations required:

read(push(n0,L) = n0

read(down(push(n0,push(n1,L)))) = n1

read(down(down(push(n0,push(n1,push(n2,L)))))) = n2

Enumerating all the required equations is not particularly enlightening for the implementor and is a gross violation of Liskov and Zilles minimality criteria. To solve this problem, hidden operators are introduced [Ref. 7]. In Majster's stack, Fasel proposed using a hidden operator "append". Append adds a new element to the top of the stack without changing the current position. If the current position is at the top of the stack then append has the same effect as push followed by down:

append(n,new) = down(push(n,new)

append(n2,push(n1,L)) = down(push(n2,push(n1,L)))

If the current position is other than the top, then axioms would show that appending is unattached to the down operator:

```
append(n,down(L)) = down(append(n,L))
```

The read operator can then be defined in terms of the read and push operators:

```
read(push(n,L)) = n
read(append(n,L)) = read(L)
```

Append would not be an operator available to the user. Its creation was to allow a finite specification of an otherwise infinite specification. Hidden operators like append would be declared under the header HIDDEN to alert the specifier of its special use.

Hidden operators present problems when overused. In particular, they suggest an implementation [Ref. 5]. The append operator is an example of this. The read operator problem may be solved in ways other than using append that is less suggestive in an implementation.

In Speclang, hidden operators are declared under the "HIDDEN" header.

3. Axiom Part

The axiom part implicitly describes the behavior of the previously declared operators. It is the most difficult portion of a specification to construct; developing the axioms to reflect only the desired behavior and no more is tricky.

In Speclang, the equations which define operator behavior are placed below the header "AXIOM". The variables used in the operator are assumed to be of sort types used in the declaration of the operator. The format for axioms is as follows:

```
SPEC <spec_id> IS
    SORTS
        sort1;
    OPERATIONS
    PRIMITIVE
        op1 : sort1-> sort1;
AXIOMS
    op1(x1) = x2;
```

The equations are composed of a left side and a right side separated by an equal sign. The equal sign does not mean the two sides are equal. It signifies that the terms that are produced from the operations on each side are in the same equivalence class. Although this is an important concept for the underlying theory of algebraic specifications, it is not vital to the person developing a specification. Another way of viewing equations is that each operator is defined by an equation that shows what happens when the operator is applied. For example, in defining the "push" operator on a stack S, we would want to show that for any element e, push(S,e) produces a stack with e on top. The specifier should know that the top of a stack is also related to the "pop" operator. The solution then in defining "push" is to relate it to the "pop" operator as follows:

$$pop(push(e,S)) = e$$

The implicit definition of behavior through algebraic axioms provides independence from implementation but also creates problems in constructing a specification.

4. Speclang Modifiers

When developing a complex program, a programmer will modularize functions. The benefits are a more readable program and the program is easier to develop and maintain. An algebraic specification using Speclang also can be developed this way.

Burstall and Goguen [Ref. 8] presented a structured language which effectively breaks a specification into smaller more manageable units. The language uses "theory_building" modifiers which permit the specifier to modify previous specifications. Fasel used their theory-building modifiers to develop his specification of a program. The proposed modifiers to accomplish this were combine, induce, extend and derive. Speclang uses only the extend modifier to create specifications.

The extend modifier adds new operators, equations and sometimes sorts to a specification. Furthermore, the extend modifier allows the specifier to combine two or more imported specifications to create the new specification. When two or more specifications are combined, all of their sorts, operators, and axioms are lumped together to form the new specification; this method is a shorthand way of defining the sorts, operations, and axioms which minimize the specification.

In order to show the use of the extend modifier without combining, Fasel's example of the Natural specification is presented:

```
SPEC Natural IS
     SORTS
        nat;
     OPERATIONS
        zero:-> nat;
        succ: nat -> nat;
```

We can extend the specification with addition and multiplication to create a new specification Natplus:

```
SPEC Natplus IS
     EXTEND
         Natural;
     WITH
        OPERATIONS
            PRIMITIVE
```

```
                add: nat,nat -> nat;
                mult: nat,nat-> nat;
        AXIOMS
            add(n,0) = n;
            add(succ(m),n) = succ(add(n,m));
            mult(0,n) = 0;
```

The extend modifier draws in all the operators, axioms and sorts from the specification name following extend and in this case from Natural. Additional sorts, operators, and axioms are included following the heading "WITH".

If the extend modifier was used to combine specifications as well as include new sorts, operations, and axioms, then Natplus could have been extended with specifications Boolean and Natural to produce the following specification:

```
SPEC Natplus IS
    EXTEND
       Boolean;
       Natural;
    WITH
       OPERATIONS
          PRIMITIVE
              add: nat,nat-> nat;
              multiply: nat,nat -> bool;
              equalto: nat,nat -> bool;
              if_then_else: bool,nat,nat -> nat;
       AXIOMS
           add(n,0) = n;
           add(succ(m),n) = succ(add(n,m));
           multiply(0,n) = 0;
           multiply(n,succ(m)) =
                       add(n,multiply(n,m));
           equalto(0,succ(m)) = false;
equalto(0,0) = true;
equalto(m,m) = true;
if_then_else(T,v1,v2) = v1;
if_then_else(F,v1,v2) = v2;
```

In this case we have two specifications, Boolean and Natural, which are used to define the Natplus specification. All of the specifications declared by the identifiers between "EXTEND" and "WITH" are combined to produce the new specification. In addition, new sorts, operators and equations are defined on the combined specification, and are declared following "WITH".

Although the Natplus specification could have been easily defined without the extend modifier, a larger, more complex specification would be difficult to read and probably impossible to understand. For example, the abstract machine specification described in chapter I has one specification module which represents the final specification. Its extend modifier combines four other specification modules and extends them with approximately one hundred other operators and seventy equations. If this machine was described without the extend modifier, then each of the combined specification's sorts, operators, and axioms would have to be included in the specification. Since each of the combined specifications were also extended, each of the specifications imported to create them would also be included. The result would be a specification containing hundreds of operators and equations, making the specification incomprehensible.

Before a specification can use another specification in the modifier, the specification being imported must have already been declared. In the above example, Natplus, Boolean and Natural, would have already been declared in the complete specificaticn.

5. Parameterized Modules

One method of minimizing a specification is through the use of parameterized modules. Goguen suggested this technique and Ganzinger [Ref. 9] explored it in depth. The basic concept is to use a procedure like specification to create a new specification. This is a particularly useful technique when many specifications tend to be repeated such as in the case of a list of integers, characters, real numbers, etc. Instead of a specification for each type of element, a generic template would be used to describe how a list of a particular type should behave. The parameterized specification for lists can be invoked by passing the specification for reals, integers, characters, etc. producing an instantiation of the parmeterized specification which is a new specification.

Many issues are still unresolved in parameterized specification which will not be addressed here [Ref. 9]. Before using parameterized specifications these issues should be understood.

The parameterized module consists of two primary parts. These are the parameter block and the specification block. The parameter part of a specification defines what may come inside a parameterized module during an instantiation. An instantiation is the passing of an actual specification to a parameterized specification.

The parameters can be viewed as the interface to the outside module which is invoking the procedure. It consists of sorts, operators, and axioms and is announced by the header "PARAMETERS".

The specification part is like the specification body used in primitive specification modules. It consists of sorts, operators and axioms and is declared by the header "DEFINED BY".

When the parameterized specification is invoked the formal parmeter's sorts, operators and axioms are replaced by the actual parameters passed from the designated specification.

In Speclang, the format for a parameterized specification would appear as follows:

```
          SPEC <identifier> IS
              PARAMETERS
                 SORTS
                    <sort_body>;
                 OPERATIONS
                    <op_body>;
                 AXIOMS
                    <axiom_body>;
              DEFINED BY
                 SORTS
                    <sort_body>;
                 OPERATIONS
                    <op_body>;
                 AXIOMS
                    <axiom_body>;
```

Ganzinger illustrated parameterized specifications by using the list example:

```
SPEC Lists IS
   PARAMETERS
      ENRICH
         Boolean;
      WITH
         SORTS
            elem;
         OPERATIONS
            PRIMITIVE
               equal: elem,elem -> bool;
```

```
           if-then-else: bool,elem,elem -> elem;
      AXIOMS
         equal(e,e) = true;
DEFINED BY
   SORTS
      list;
   OPERATIONS
      PRIMITIVE
         lempty: -> list;
         isempty: list -> bool;
         conc: elem,list -> list;
         cdr: list -> list;
         if-then-else: bool,list,list -> list;
         first: list -> list;
    AXIOMS
       cdr(lempty) = lempty;
       cdr(conc(e,l)) = l;
       cdr(if_then_else(b,l1,l2))
             = if_then_else(b,cdr(l1),cdr(l2));
       isempty(lempty) = true;
       isempty(conc(e,l)) = false;
       isempty(if_then_else(b,l1,l2)
          = if_then_else(b,isempty(l1),isempty(l2));
       first(lempty,e) = e;
       first(conc(e,l),e') = e;
       first(if_then_else(b,l1,l2),e)
       = if_then_else(b,first(l1,e),first(l2,e);
```

This parameterized specification is invoked by using its name and brackets around the specification which is to be used for the instantiation. Therefore, if a specification for a list of natural numbers was desired, Lists(Natural) would invoke the parameterized specification for list. In order for Lists to be correctly invoked, the Nat specification must "match" the formal parameters of Lists. In Speclang, the sorts are matched under the header "ACTUAL SORTS" and the operators are matched under the header "ACTUAL OPERATIONS". Under each of these headers the sorts and operators that are to replace the formal parameters are positioned to the left of an "IS". The formal parameters the actual parameters are replacing are positioned to the right of the "IS". In Speclang, the matching parameters are announced at invocation. Using the natural numbers invocation, it would appear as follows:

```
Lists(Natural)
   WHERE
      ACTUAL SORTS
          nat IS elem
          bool IS bool
      ACTUAL OPERATIONS
          equal IS equal
          if_then_else IS if_then_else
          and IS and
          not IS not
```

Note that the parameterized specification has an extend field in the parameter block. When this occurs, the parameterized specification must be invoked with sorts and operators that also match the specifications that were imported by the extension. In the List example, the sorts and operators in the Boolean specification were matched in the invocation.

IV. SPECIFICATION EDITOR

In order to facilitate writing specifications, a syntax directed editor was designed to format a specification created from Speclang. The editor is based upon ideas from Davis [Ref. 10] and MacLennan [Ref. 11]. It is table driven in that a grammar is input and parsed into seperate production rule trees. The production trees are then used to create a specification tree. By using the table method, flexibility is gained in altering the grammar without affecting the editor. MacLennan's editor and the idea presented by Davis was to create an internal tree from the grammar which was annotated; the tree was in a parsed form which could be used to generate the machine code. The Specification Editor presented here was designed to create a syntactically correct specification and to make the interface with the user simple and friendly. The internal tree created by the editor is not annotated as prescribed by Davis and MacLennan. The editor could be modified to incorporate annotations which could transform the syntax tree created by the tree to an annotated tree. A grammar grammar was developed for the parser which is based on the modified BNF notation presented in chapter III.

The grammar used in the editor is the same as that in appendix A except for the following:

1. Comments have not been incorporated.
2. The input grammar cannot have selection involving expressions. For example, the following production rule would not be acceptable since the right hand side contains an expression with the selection:

```
<spec_block>-> (<param_call><cr><indent>)|<spec_body>)
```

This restriction was imposed for two reasons; to force a clear display selection and to reduce the problems involved in programming the recursive descent parser. This restriction is not severe since the right hand side could be replaced with a selection such as <param_call> creating two rules:

```
<module> -> (<param_call>|<spec_body>)
<param_call> -> <call_block> <cr> <indent>
```

3. The rules were modified to reduce the number of selections available. For example, the rule for a <module_spec> is as follows:

```
<mod_spec> -> <spec_header><modifiers>?<spec_body>
```

This was combined with the rule for <spec_header> to create a rule:

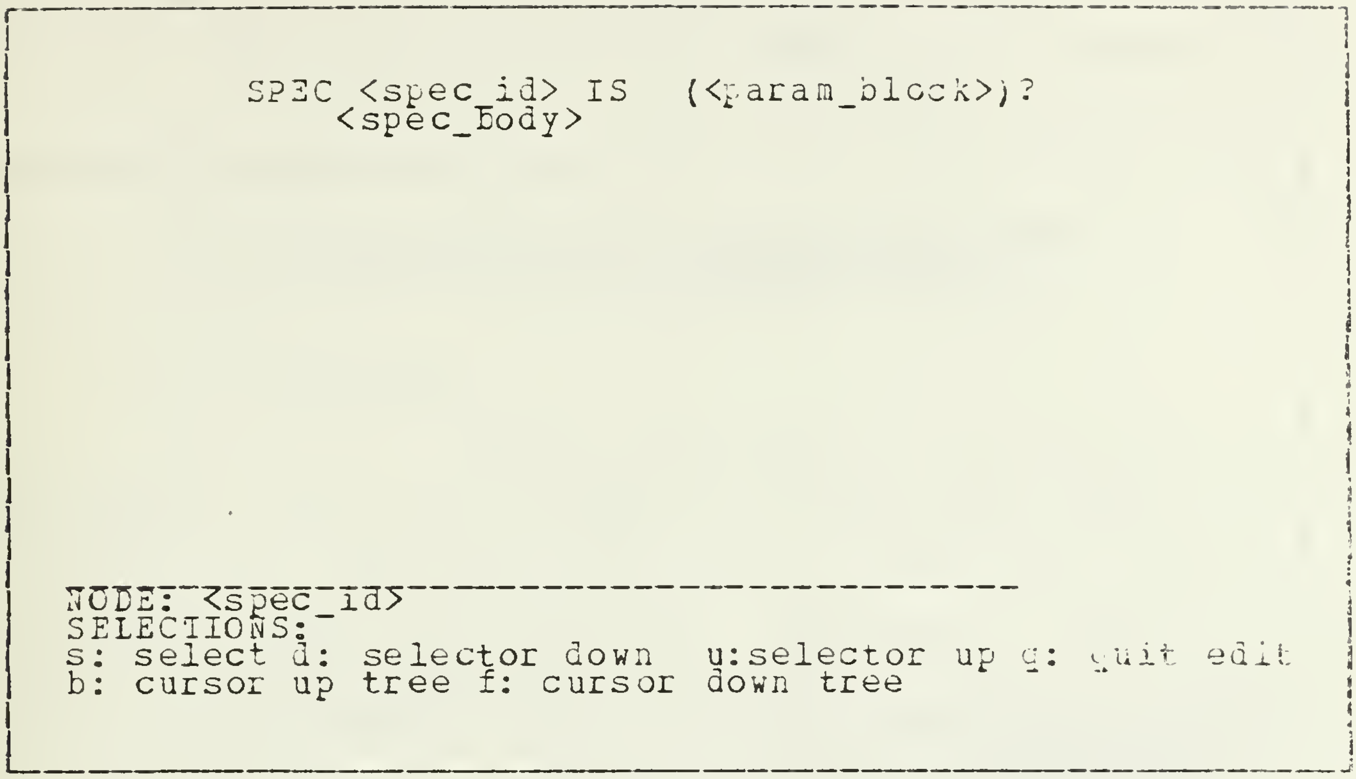
```
<module_spec> -> 'SPEC' <spec_id> 'IS' <indent>
           <cr><modifiers>? <spec_body>
```

A. AN EDITING SESSION

This section will describe an editing session. The editior is initiated by entering the name of the editor, Speced. A prompt appears requesting the name of the file where the user had stored a tree from a previous editing session.

The screen is cleared and the first production rule's right hand side is displayed on the screen in an "unparsed" form. The video is then reversed on the first nonterminal or modifier which an input selection can effect. All of the strings which are displayed in reverse video are referred to as the selector field.

For example, if <module_spec> was the first rule, then the display would be as depicted in Figure 4.1 .



```
          SPEC <spec_id> IS  (<param_block>)?
              <spec_body>







  NODE: <spec_id>-----------------------------------
  SELECTIONS:
  s: select d: selector down  u:selector up q: quit edit
  b: cursor up tree f: cursor down tree
```

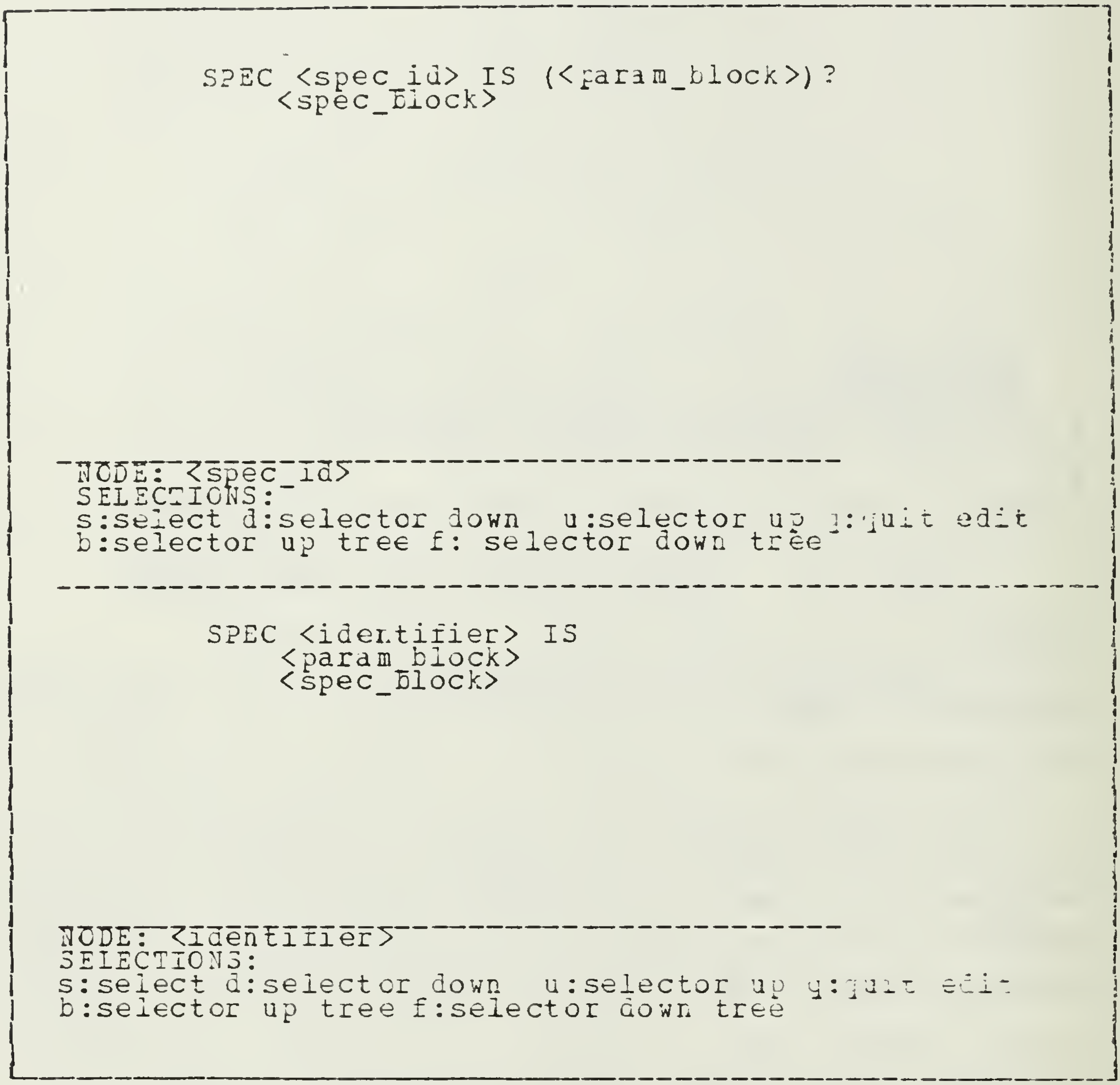
Figure 4.1 Display of the Rule <module_spec>.

In this instance, the video for <spec_id> would be reversed denoting the selector, and the bottom of the display would show the selections available.

The selector is directly related to a selection pointer, sel_ptr, which points to a node in the specification tree being edited. The node at which the sel_ptr is pointing is referred to as the current node. The current node is shown at the bottom of the display following "NODE:". Depending on the current node a number of different selections are available. The possible selections a user can make during an editing session are described in the following sections.

1. "s" Selection

If "s" is selected and the current node is a nonterminal, the node is replaced by the production rule identified by that nonterminal. In the above example, if <spec_id> is selected, then the rule for <spec_id> is inserted into the abstract specification tree, the screen is updated, and the selector is advanced.



```
          SPEC <spec_id> IS (<param_block>)?
              <spec_block>









   NODE: <spec_id>
   SELECTIONS:
   s:select d:selector down  u:selector up q:quit edit
   b:selector up tree f: selector down tree

-------------------------------------------------------------

          SPEC <identifier> IS
              <param_block>
              <spec_block>









  NODE: <identifier>
  SELECTIONS:
  s:select d:selector down  u:selector up q:quit edit
  b:selector up tree f:selector down tree
```

Figure 4.2   Selection of a Non-terminal.

If the current node is an option node, then the "s" selection will inhibit the display of the question mark and the nonterminals/terminals preceeding the modifier will become part of the abstract specification tree. A similar result occurs when the current node is a selection modifier. In this case, all selections which are no longer needed along with the selection symbols are not displayed when the desired selection is made.

Figure 4.2 shows the tree before and after an update where <spec_id> and <param_block> were selectel.

2. "i" selection

The "i" or insert selection is available only when the current node is an identifier nonterminal designated by <identifier>. When this selection is made, the portion of the screen where the identifier occupied is blanked. The user at this point can then type in the identifier desired. The input characters are then checked to ensure correct lexical grammar for identifiers. If the input character is incorrect, it will be ignored and the eiitor will wait for another input. The input string can be terminated with a S input. In any case the string will be terminated when its length exceeds twenty characters. After terminating the input the editor updates the screen and advances the selector.

3. "b" and "f" Selections

The "b" and "f" inputs move the selector up and down the screen respectively.

Some nodes the sel_ptr stops on are not displayable. In these cases, all the descendants in the tree that are eligible are displayed via the selector. As an example, if <module_spec> is the current node then all of the displayel characters in figure 4.2 would be inverted and <module_spec> would be displayed at the bctton of the screen after "NODE:".

4. "u" and "d" Selections

The "u" and "d" selections are similar to the "b" and "f" selections; they move the selector up and down the screen. The difference is that the "u" and "d" selections will move to the next displayable node on the screen. This facilitates moving the selector on the screen.

5. "m" Selection

The "m" selection is used when the current node is a selection modifier node. Since the "f", "b", "u" and "d" selector movement selections will not allow moving the selector into a modifier node subtree, the "m" or "make selection" key was created so the user could move the selector over the desired selection. If the selector is on the last selection and "m" is selected, then the selector will move back to the first selection. The selector will remain in the selection tree until either a selection is made or the user makes a "j" selection to jump out of the selection.

6. "e" Selection

The "e" selection allows the user to "erase" the string the selector is on. This means that the string will no longer be displayed on the screen. The "e" selection is available when the current node is the option node or a nondisplayed nonterminal.

If the selector has an option symbol in its field, the erase selection will eliminate the option symbol, brackets if displayed, and the nonterminals and modifiers in the selectors field. In figure 4.2 if <param_block> was not desired and the selector was positioned on (<param_block>)?, then the erase selection would yield the following display:

```
SPEC <identifier> IS
    <spec_body>
```

When the node indicated at the bottom of the display is not a displayable node, the erase will eliminate all of the strings of the screen which are in selector's field and replace them with that node.

7. "r" Selection

The "r" selection is used to replace an option node in the display or a selection node. In the previous example for erase, if the sel_ptr was positioned in the specification tree so that the option may be reselected, then the "r" selection would replace the option tree in the specification tree and the display would be updated accordingly. A selector would not be visible in the case where the option was previously erased. The replace field at the bottom of the screen would indicate the option to be inserted into the tree.

In the case that the cption or selection has been previously made, then the entire portion of the tree from the option or selection would be eliminated. The strings that would be eliminated are in the selector's field. For example, if the specification was as follows:

```
SPEC <identifier> IS
     PARAMETERS
         <spec_block>
     DEFINED BY
         <spec_block>
```

and the current node was the "not" displayed option node <param_block>?, then a "r" selection would yield:

```
SPEC <identifier> IS (<param_block>)?
     <spec_block>
```

B. DESIGN OF SPECED

The editor was written in Pascal and implemented on a Digital Vax 11/780 under the VMS operating system. The editing session described in the preceeding section was the objective before the design of the editor started.

The design of the program was done in a top-down manner. The three primary modules are as follows:

1. Initialization
2. Main body loop
3. Termination routine

1. Early Design Decisions

In order to simplify the display, certain nonterminals are treated in a special manner. The <indent> and <cr> nonterminals are immediately interpreted by the editor. These nonterminals mean indent and carriage return respectively.

The indent nonterminal, <indent>, before a nonterminal or expression means that the nonterminal and its descendants will be indented by a preset tab or spaces if a carriage return follows.

The identifier nonterminal, <identifier>, is treated as a slot for inputting a string of characters. This seems reasonable since all grammars require identifiers. The lexical composition of the identifier is built into the editor.

2. Initialization

The first routine called by the main program is Initialization. This routine in turn calls a routine called opengrammar to input the production rules from a grammar file. Opengrammar has a scanner that preprocesses the input rules to ensure that lexically they are correct. It "looks" for a bracketed left hand side of each rule , an "->", and then bracketed strings, quoted strings and metasymbols. The grammar rules are put into a record structure which consists of a left hand side and a right hand side of the production rules.

The next procedure called is parsegrammar. This routine is a recursive descent parser. The parser is based on a LL(1) grammar grammar used to construct the production rules. The grammar grammar for the parser is as follows:

```
<production_rule> -> <nonterminal><rule_body>
<rule_body> -> ((<nonterminal>|<expression>)('*'|'+'|'?')?)
<expression> ->'{'(<rule_body>|<ident_list>)'}'
<ident_list> -> <nonterminal> ('|'<nonterminal>)+
```

Each production rule in the grammar grammar is mimicked by a module in the recursive descent parser. Every production rule in the grammar for Speclang is parsed into a separate production rule tree which describes its syntactic structure.

Figure 4.3 shows an abstract production rule tree for the <module_spec> production rule.

For each production rule tree there is pointer in an array of pointers to locate that rule tree. The rule trees are located by searching the array for the root with a name that matches. A production rule tree is constructed from records which have pointer and integer fields as follows:

```
grammrules = record
                 parent:array(1..20) of character;
                 len:integer;
                 child:array(1..30) of |grammrules;
                 par_ptr:|grammrules;
                 ch_no:integer;
                 rtype:integer;
                 num_ch:integer;
                 disp:boolean;
                 ln: integer;
              end
```

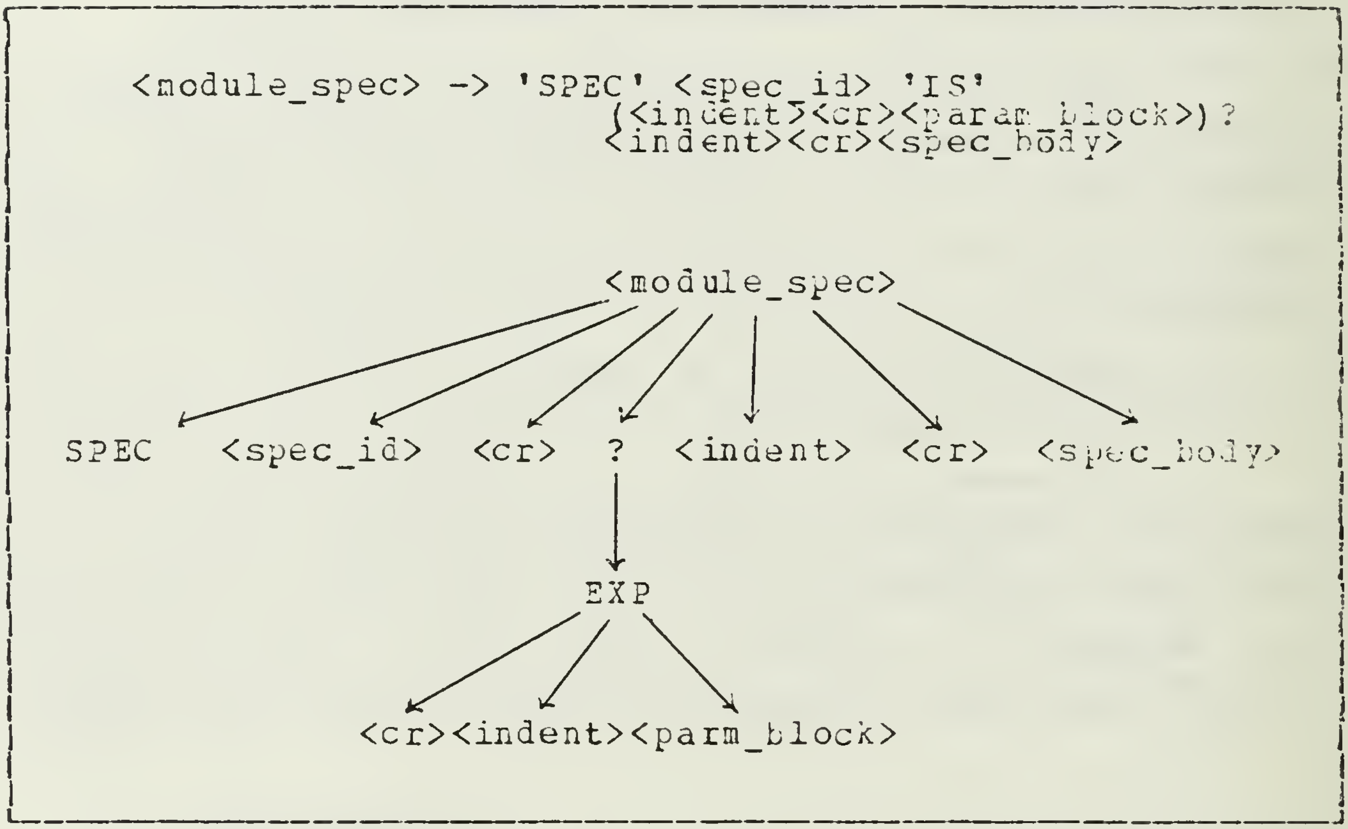


Figure 4.3 A Production Rule Tree.

The "parent" identifies the name of rule, string, or modifier. The field after parent, "len", contains the length of the parent character string. This feild is used in outputting the string on the screen. The "child" array points to the descendants of that rule. A pointer called "par_ptr" points to the parent of that node. This pointer was necessary in order to move the selection pointer around the abstract tree. The "ch_no" field identifies what child number that record node is of a parent record node whereas the num_ch field indicates the number of children that nodes has. The ch_no and num_ch fields were also created to facilitate selector movement. The "ln" field is used for the display screen. field tags the "type" of record node. There are seven types of nodes as shown in Table I .

TABLE I

Node Types

Type	Description
1	nonterminals
2	modifiers-?,*, ,+
3	terminal strings
4	expression node
4	identifier strings
6	ALT node
7	ignore node

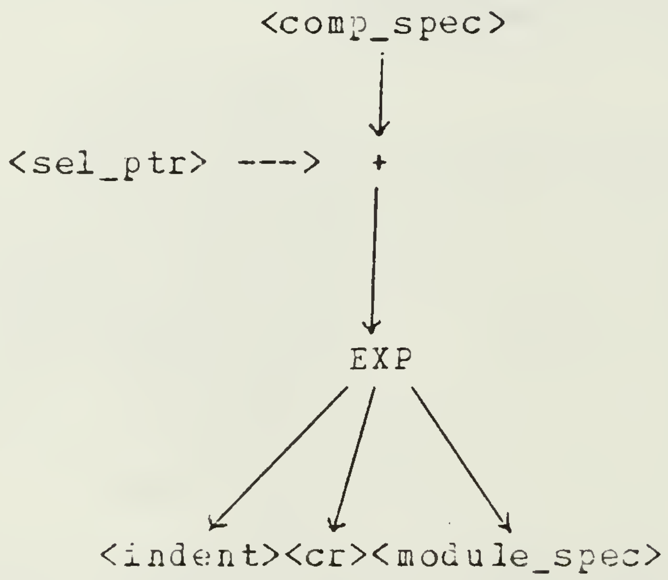
The node type has an effect on the selections available, how the tree is "unparsed" for display, and the movement of the selection pointer.

After the rules are parsed a procedure, get_tree, will open the file of a specification tree that was stored after a previous editing session, if the user inputs the name of the file. If a user stored tree is not supplied, the initialization routine will copy the first production rule tree and use it as the root of the specification tree.

The final steps of the initialization routine set up the display for an editing session. A pointer, prog_ptr, will be set to point at the root node. If a user specified tree is available then the prog_ptr is set to the root of that specification tree.

The next step is to clear the screen and call a routine, print_tree, which "unparses" the tree to produce the display on the screen.

The final step of the initialization routine assigns the selection pointer to the first selection of the tree. If the initial production rule was displayed the abstract tree would appear as follows:



a. Print Routine

The print routine, which unparses the specification tree being developed and displays it on the screen, deserves further explanation due to its complexity. It is a recursive procedure which does an inorder traversal of the specification tree and, as it visits each node, will respond according to that node's type.

Initially, the prog_ptr is passed to the print routine and then a case statement determines how to handle the root node and all nodes recursively passed to it. Since the root node is the left hand side of the first production, it is not displayed. Instead, it treated as a nondisplayed nonterminal node which is explained below.

The predominant factor as to whether or not a node parent's field is to be displayed is the record boolean field "disp". Only if the field is true will it be output to the screen.

Another factor for displayability is the line number of the node. Since the screen can only display a small fixed number of lines of the specification at one time, a current line is assigned for the display which will dictate what nodes in the specification will be shown. If a node's "ln" field is in a particular value range of the display line number then it will be displayed. Otherwise, even if the disp field is true, it will not be displayed. For example, if the current line is twenty, then all the specification nodes whose ln field falls in the range twenty to thirty five will be displayed.

As mentioned before, the print routine responds to a node's type via a case statement which has seven cases that correspond to the seven node types in Table I.

Case one handles nonterminals. This case checks initially to see if the nonterminal is a carriage return or an indent. If it is a carriage return, the screen line number is incremented and the selector positioned on the next line of the screen. If the nonterminal is an indent symbol, a global indent counter is incremented by one and an indent flag is set. This mechanism is used to cause the nonterminal and its descendants following the indentation to be indented.

Before a nonterminal or its descendants are displayed on the screen, a global flag is checked to see if the preceding nonterminal was an indentation. If it was, then the global flag is set to off and a local flag is turned on. After the nonterminal or its descendants are displayed, the local flag is checked. If the flag is on, the number of immediately preceeding indentations are subtracted from an indentation counter and the local flag is turned off. Using this method, the indentation affects only the nonterminal or the expression following it. For example, if the rule for <module_spec> was:

```
<module_spec> -> SPEC <spec_id> IS
                       (<indent><cr><param_block>)?
                        <indent><cr><spec_block>
```

The initial display would be as follows:

```
SPEC <spec_id> IS (<param_block>)?
   <spec_block>
```

When <param_block> is expanded the indentation will affect all of its descendants so that the display would appear as follows:

```
SPEC <spec_id> IS
    PARAMETERS
       <spec_block>
    DEFINED BY
       <spec_block>
```

Note that <spec_block> is indented by two. This happened because the rule for <param_block> has an <indent> at the end and the rule for <module_spec> has an <indent> preceeding it. If <param_block> was eliminated, then <spec_block> would only be indented by one.

If the node passed to the print routine is not a carriage return or indent, then the boolean field "disp" is tested for displayability. If the field is true then the node's parent field is written to the screen. If the nonterminal is not to be displayed, then a recursive call is made to the print routine for each of the pointers in the "child" field of that node.

Case two handles all the modifier nodes. If a selection modifier node is encountered and is to be displayed, the print routine will display all the selections and place the "|" symbol between the choices. All the other modifier nodes will have a recursive call to the print routine on their descendants. After returning from the recursive call, the node is checked to see if it is to be displayed. This is how the postfixing of the modifier symbols is accomplished.

Case three handles the template nodes. These are nodes that have strings that are required in the grammar. For example, every module must have the string "SPEC" and the string "IS" on the first line. These nodes have no descendants and so the print routine will not make any recursive calls on the descendants.

Case four deals with expression nodes. An expression node is used to point to the descendants; its only function is to allow the print routine to make recursive calls on each descendant. If a global flag is set in the case two condition indicating the expression is followed by a modifier, then brackets surrounding the expression will be displayed.

Case five prints the identifier nodes. These are terminal strings for identifiers and are treated like the case three nodes.

Case six deals with the "ALT" nodes. This node is used to store trees such as options and selections after these nodes have been acted on. It also retains the Kleene and alternation trees. Normally the ALT node has two descendant nodes. The left one is for the branch leading to displayed strings. The right one is for storing the option, selection, Kleene and alternation trees or it may point to another ALT node. The print routine will recursively call each of the children nodes.

Case seven is the ignore case. This is used when the node and its descendants are to be ignored. For example, in the case where the option node was used and stored. The option expression is no longer desired for display so it is given a type seven.

3. Main Body Loop

The editor, after initialization, enters a loop in the main program body which inspects the type node at which the selection pointer is pointing and calls one of eight routines to handle that case. The routines are shown in Table II .

All of the routines which handle the various node types are essentially constructed the same. A procedure, sho_slection, is called in each routine which displays the various selections available in that routine at the bottom of the screen. Each routine then enters a loop which retrieves the input from the terminal and deciphers the input via a case statement. If an invalid character is selected, it is ignored and the routine loops for another input.

The following sections will discuss the routines in Table II with the exception of the first section which discusses the selector movement routines.

TABLE II

Routines called by Main Body Loop

Node Type	Routine
Nonterminal	Id_replace
?	Option
|	Selection
+	Alternation
*	Kleene
identifier	Id_deselect
ALT	altrep

a. Selector Movement Routines

There are four routines used to position the selection pointer on a node in the specification tree in concert with the selector on the screen. They each correspond to one of the four selections, "u", "d", "f", and "b", available in all the routines shown in Table II . When "d" or "u" inputs are received, one of two routines will be called to move selection pointer and the selector to the next displayable selection either down or up the screen; this means the selection pointer will be at a node that is visible on the display.

The "b" or "f" inputs move the selector backward or forward to any node in the specification tree that is available for modification. These two inputs also have two separate routines which move the selection pointer in the tree. The "b" and "f" selections allow the interior nodes, which are not displayed on the screen, to be available for alteration.

All of the selector movement routines are designed so that they will not position the selection pointer on "EXP" nodes, modifier nodes ,if they have been erased or selected, and terminal type three nodes that contain strings usei in the grammar. The nondisplayed modifier nodes are accessed in a different manner. After the selection pointer is moved, the current node is checked to determine if it is on the screen. If it is not, the current screen display line is updated to position that node in the center line of the screen. The print routine is then called to update the screen.

b. Replace Routine

The replace routine acts on type one nonterminal nodes. Two general cases are possible, depending on whether the node is displayed. In the case where the nonterminal is displayed, the replace routine enables the user to add to the specification tree the production rule tree indicated by the nonterminal node's parent field. As mentioned in the previous section, this occurs when an "s" is input. Figure 4.4 shows the specification tree before and after an "s" is input and the selection pointer is at <spec_id> in a specification tree.

When "s" is selected, a routine called clone locates and reproduces the the production rule tree for <spec_id> and returns a pointer to the replace routine. The replace routine then attaches the rule tree to the specification tree. The <spec_id> "disp" field is then set to false to inhibit its display.

It is possible that the selection pointer is at a type one node that is not displayed; its disp field is set to false. In this case, the displayable descendants are shown in the selector's field. An option to erase is then available to eliminate all the descendants and restore the

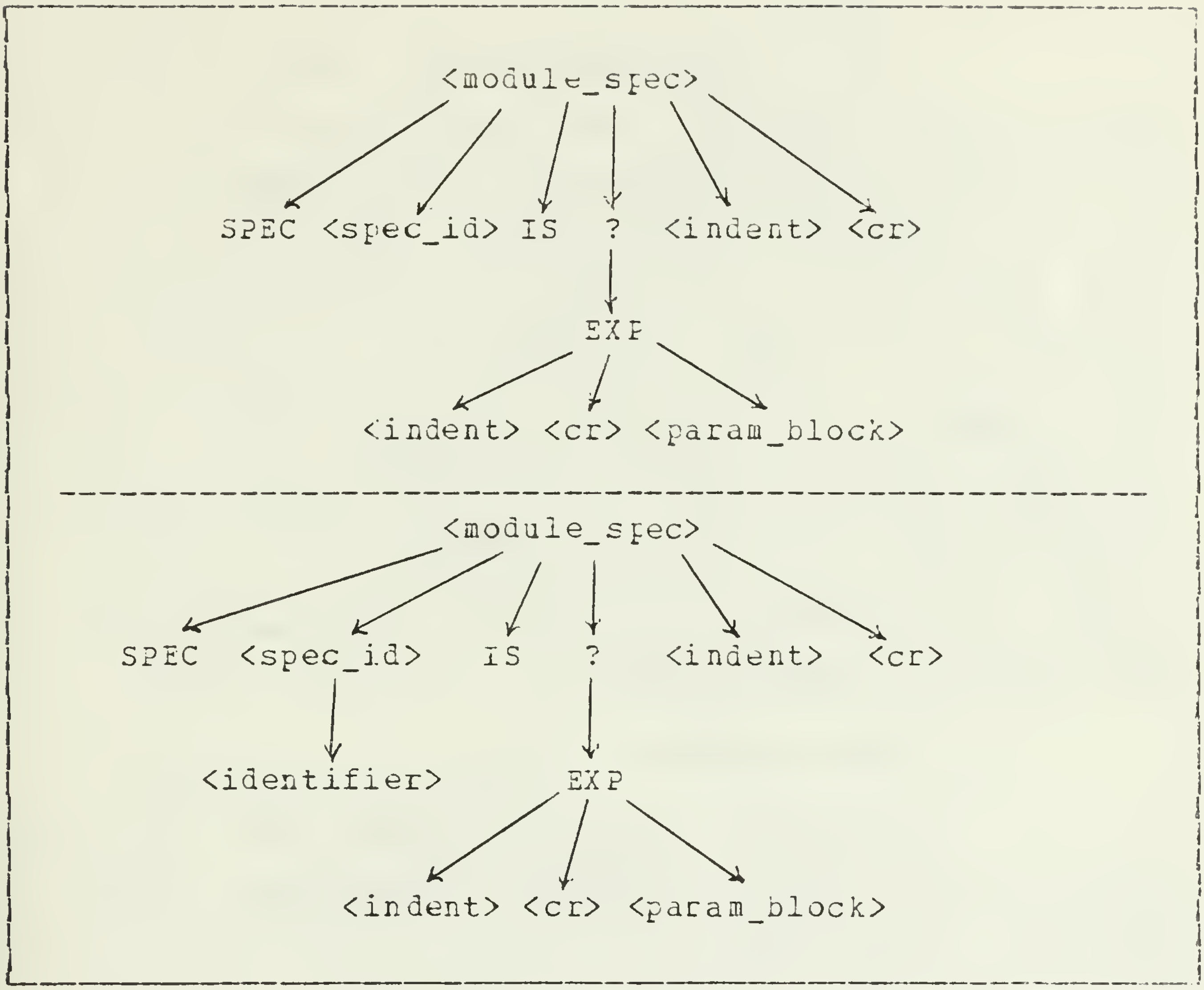


Figure 4.4 Selection of <spec_id>.

not displayed nonterminal. This is accomplished by setting the current node's child pointer to nil and the disp field to true. The "NODE" field at the bottom of the screen shows the not displayed nonterminal.

One special case is addressed in this routine. This is the case when the selection pointer is at the root of the specification tree. In this case, the first production rule tree is duplicated and substituted when the "e" selection is made. This is paramount to erasing the entire tree.

c. Id_replace

The id_replace routine was created to allow the user to input directly on the screen the identifier names. This routine is called when the selection node type is one and the nonterminal is <identifier>. When an "i" is input, a case statement is executed which determines the starting position on the screen by inspecting the current node's posit field. A blank is written to the screen starting at that position. The routine then takes inputs from the terminal and places them in that node's parent field. Each character input is checked for proper lexical grammar and is echoed to the screen. Invalid characters are ignored. The inputs are terminated when either twenty characters have been input or a "$" is input.

d. Option Routine

The option routine is called when the current node is type two and the parent field is the option node modifier. This routine will allow the user either to erase the optional expression by selecting an "e" or include the expression preceding the modifier by selecting an "s".

When the option is selected or erased an "ALT" node is inserted into the option node's position in the tree. The ALT node is treated as if it only has two children. The right child will store the option tree so that if the user later desires to reverse his previous decision and include it, he can reinsert the option tree into its prior position in the tree. The left child of the ALT node will be nil if the option is erased or will contain the expression that was preceding the option modifier if the option was selected. Figure 4.5 shows the tree before and after the selection of the <param_block> option.

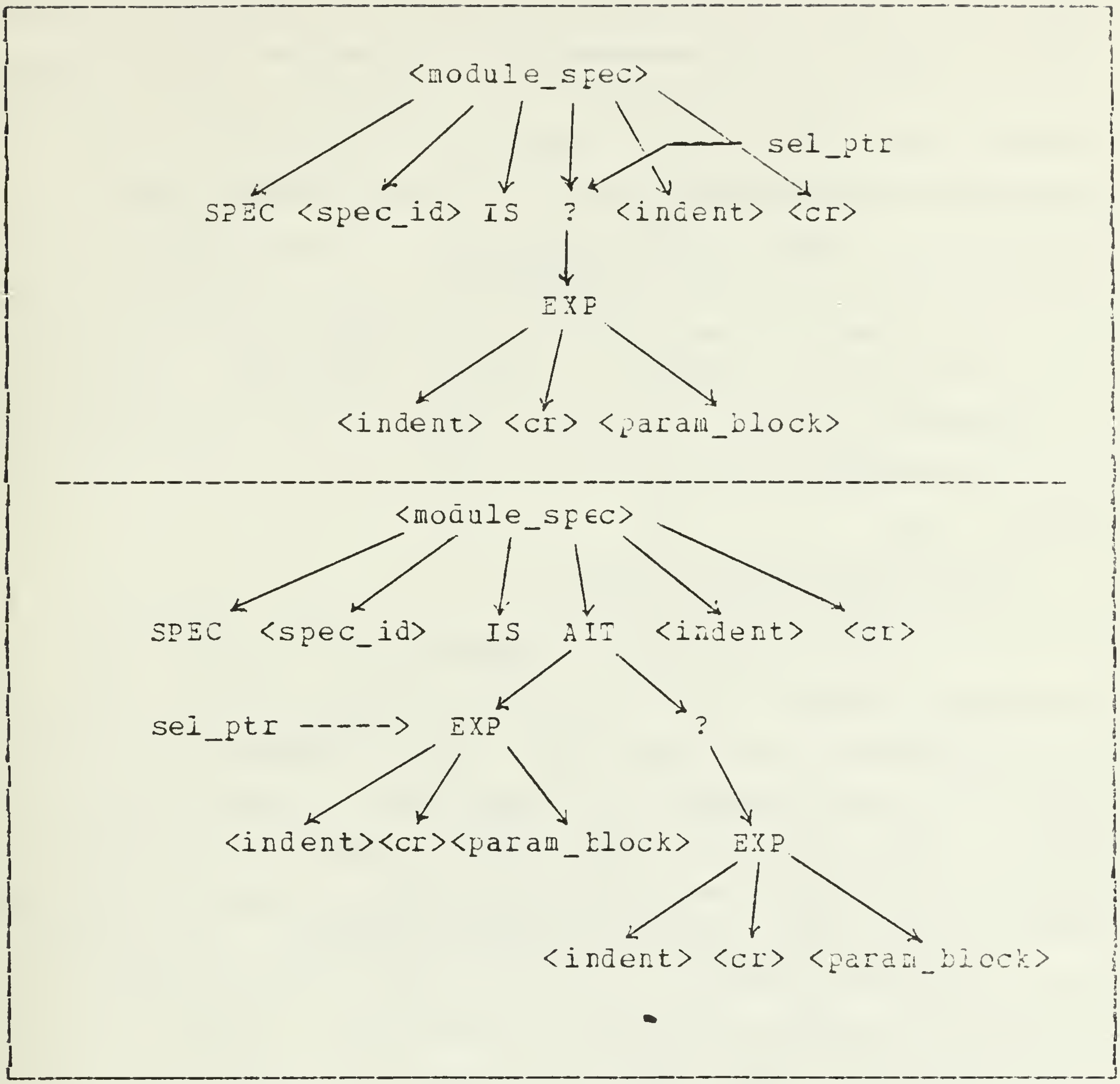

<module_spec>
sel_ptr
SPEC <spec_id> IS ? <indent> <cr>
EXP
<indent> <cr> <param_block>
<module_spec>
SPEC <spec_id> IS ALT <indent> <cr>
sel_ptr ----->
EXP
?
<indent><cr><param_block>
EXP
<indent> <cr> <param_block>

Figure 4.5 Selection of the Option <param_block>.

When the option is stored in the right child, it is assigned a type seven so that the print routine will ignore it and its descendants.

e. Selection Routine

The selection routine is called when the selection node is type two and the parent field is the selection modifier. Because the selector down or selector up routine can not access the descendants of a selection node or option node, the selection routine was written to permit the user to designate the desired string by using the "m" key to position the selector over it. When "m" is input a case statement is executed which will loop between the selections until either a selection is made or a "j" is entered to jump out of the loop. When a particular nonterminal is selected the routine behaves like the option selection; an ALT node is inserted for the selection node and the selection node and its descendants are stored in the right child of the ALT node.

f. Alternation Routine

The alternation routine is called when the selection pointer is at a type two node and the parent is the alternation modifier. The routine allows the user to duplicate the expression or nonterminal preceding the modifier. When the "s" selection is made an ALT node is inserted into the tree at the alternation position. The left child points to a cloned expression of the descendants of the alternation modifier and the right child points to the original alternation node and descendants. Figure 4.6 shows the tree before and after a selection. Unlike the option and selection routines, the right child in this case will not be typed seven since we want to allow the user to make as many selections as he would desire. After the user has made at least one selection, then the node can be erased; the alternation routine is written so that at least one selection of the nonterminal or selection must be made. This is accomplished by checking if the alternation modifier is a child of an ALT node. If alternation modifier is, then a selection must have been made and therefore the alternation tree is eligible to be erased from the display. When an "e" is input, the routine will change the alternation node's first descendant to type seven and the disp field to false.

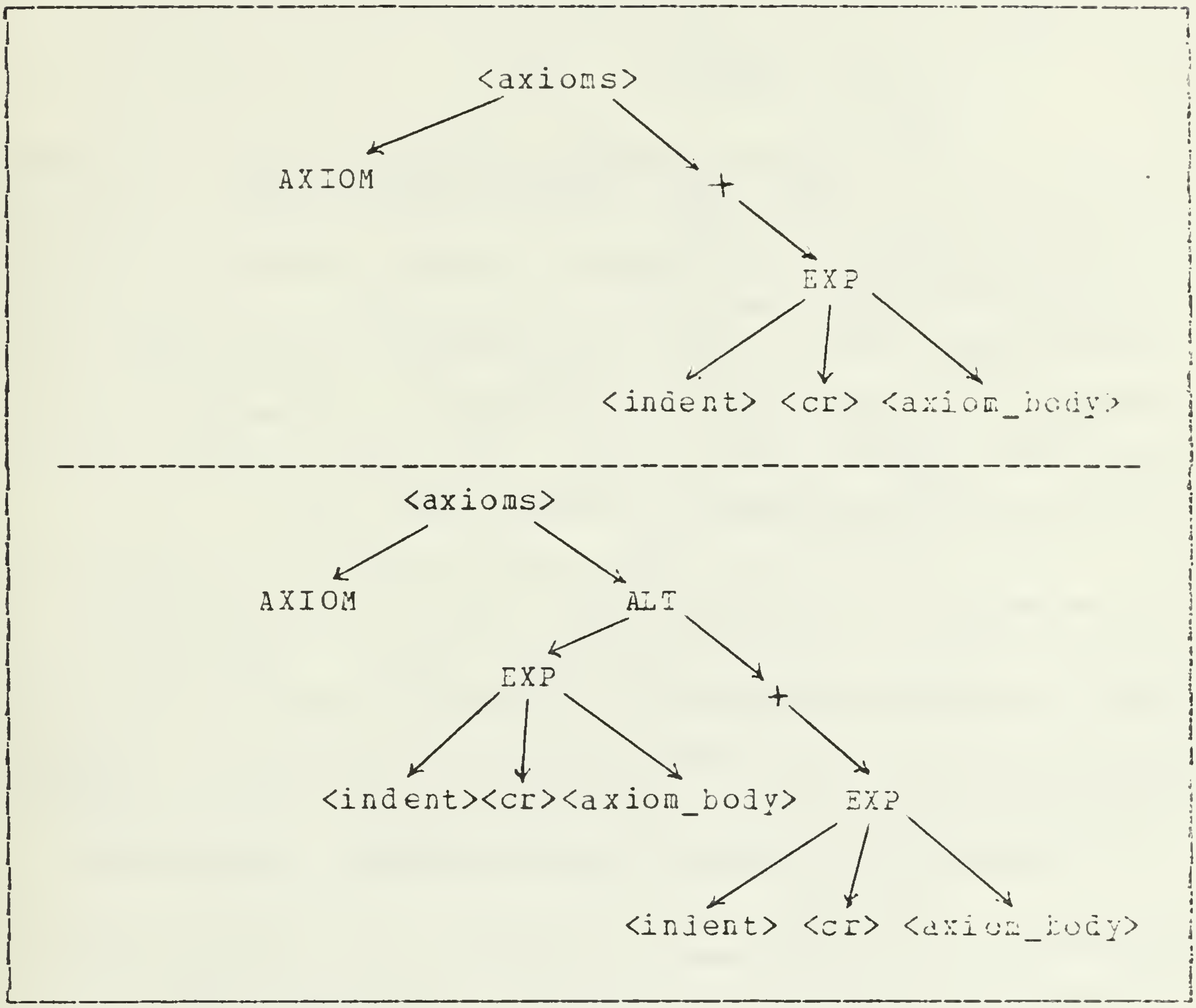


Figure 4.6 Selection of an Alternation Expression.

g. Kleene Routine

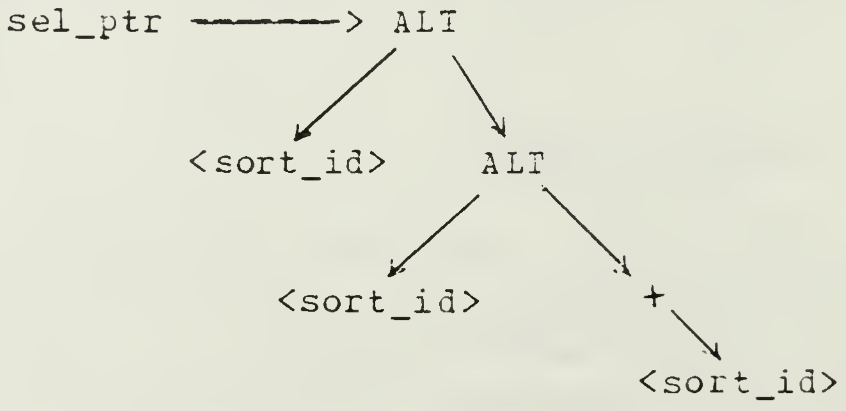
The kleene routine is executed if the current node is type two and the parent field is the kleene modifier. Like the alternation routine, it allows unlimited duplication of the descendants of the modifier. The routine is the the same as the alternation routine except that it allows the user to erase the kleene modifier and its descendants even if a selection has nct been made. To accomplish this, the kleene node's display field is set to false and the first descendant is set to type seven.

h. Altrep routine

The altrep routine is called when the selection pointer is on an ALT node. As discussed before, the ALT node is used to handle cases for the option, selection, alternation, and kleene routines. Consequently the altrep routine handles two cases; one case is when the ALT node is the result of action taken on an cption or selection node and the other case is when the ALT node is the result of action taken on an alternaticn or kleere node.

If the ALT node was the result of action taken on an option or selection node then the case statement will permit the user to erase the left tree, eliminate the ALT node and place the option or selection tree in the specification tree in its former position. Figure 4.7 shows the result of reselecting the <param_block> option which puts the <param_block> option tree in its previous position.

The case where the ALT node was the result of action on an alternation or kleene node has two subcases. Cne case is where the ALT node proceeds another ALT node such as follows:



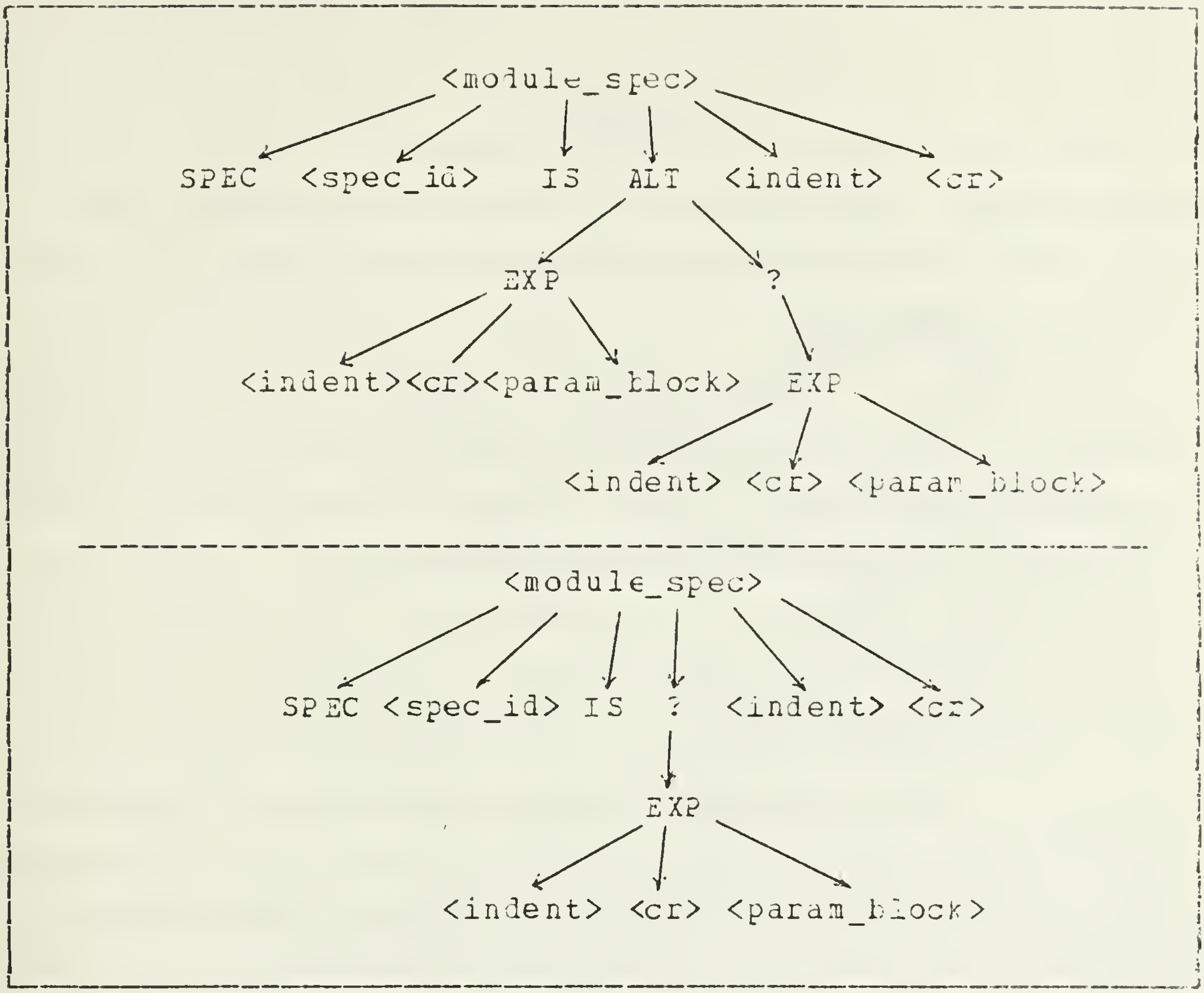
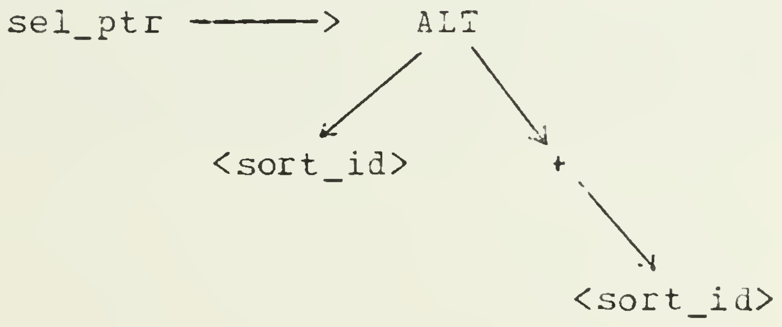


Figure 4.7 Reselecting the <param_block> Option.

In this instance, a selection is available to eliminate the ALT node, its left child and all of the left child's descendants. In this way the user may eliminate selections made before.

The other possibility is where the ALT node is adjacent to a kleene or alternation node as follows:



In this case, the selection to eliminate the ALT is available and if the kleene or alternation node has been erased, a replace selection is available to redisplay the kleene or alternation node and its descendants. This is done by setting the alternation or Kleene node's first descendant type field to seven and the disp field to true.

4. Termination

After the user inputs a "q" to quit the editing session, the editor exits the main body loop and then calls a termination routine. This routine may call two different routines depending on the users desires:

1. Store Tree
2. Pretty Print

a. Store Tree

This recursive routine performs an inorder traversal of the specification tree to store the fields of each node. It is called if the user answers yes to a prompt asking if he wants to store the abstract tree. When the routine is entered, a file is opened with the filename input at the beginning of the editing session. Each node of the abstract specification tree is then visited and its fields stored in the file.

b. Pretty Print

This routine is called when the user answers yes to a prompt asking if he desires a pretty print file of the abstract tree. This file will be an unparsed version of the abstract specification tree. It is essentially the same routine used for unparsing and displaying the tree on the screen, except this routine ignores the line number used for display.

V. SUMMARY

Speclang is a language based on algebras which has the facilities for creating formal specifications that fit the criteria established by Liskov and Zilles. Speclang reduces the complexity of a specification, and makes a specification more readable and easier to construct by using a modular hierarchy; module specifications are built by importing other specification modules through the extend modifier. Parameterized specifications modules help to minimize a specification by eliminating redundant specification modules. Futhermore, indentations and carriage returns imbedded in the grammar help to promote a consistent format between specifications. A consistent format will assist readers of the specifications developed from the Speclang grammar.

Because there are many unresolved issues in using algebras for specifications which may effect the format and grammar of Speclang, a table driven syntax directed editor, Speced, was created to assist the specifier. Its purpose is to create syntactically correct specifications. Furthermore, the design of the editor permits the user to easily modify the grammar.

APPENDIX A
SPECLANG GRAMMAR

The production rules are written in a modified Backus-Naur (BNF) notation. An explanation of the meta-symbols used to produce terminal strings in the grammar are as follows:

< > - A name enclosed by pointed brackets indicates a non-terminal in the grammar.

' ' - Strings in quotes indicate terminal strings.

( ) - Rounded brackets indicate the scope of a modifier symbol.

* - The Kleene closure modifier may follow either rounded brackets, terminals or non-terminals. It means that zero or more of the expressions, terminals or non-terminals may be produced.

+ - The alternation modifier is used like the Kleene modifier. It means that one or more expressions, terminals or non-terminals may be produced.

| - The selection modifier indicates a choice of non-terminals, terminals or expressions. It is used between two or more choices. In order to clarify the selection, rounded brackets must enclose the selection.

? - The option modifier indicates that the terminal, non-terminal, or expression is optional and can be excluded.

-> - The production symbol indicates what a non-terminal can produce.

A production rule consists of a non-terminal followed by a production symbol followed by an expression. An expression is comprised of terminals, non-terminals, modifying symbols, and may include other expressions.

The grammar for Speclang includes indentation and carriage returns to force a formatting of the specification.

A specification is complete when all the strings in the specification are terminal strings. This is accomplished by starting with a nonterminal and substituting the nonterminal which are on the right hand side of the production rule for it. The remaining nonterminals are then substituted until all strings are terminal strings. For example, starting with the nonterminal <spec_module> and substituting the right hand part of the rule results in the following:

```
          <spec_header> <param_block>? <spec_body>
```

The rule for a specification header is:

```
 <spec_header> -> 'SPEC' <spec_id> 'IS' <new_line> <indent>
```

It right hand side is substituted into the above string to produce:

```
  'SPEC' <spec_id> 'IS' <new_line> <indent> <param_block>?
                         <spec_bcdy>
```

<newline> and <indent> are interpreted as carriage return and a tab respectively, except when they are in an expression followed by a modifier, and make the developing specification appear as:

```
               SPEC <spec_id> IS
                   <param_block>? <spec_block>
```

Since <param_block> is optional it can be eliminated to produce:

```
                    SPEC <spec_id> IS
                    <spec_body>
```

The substitutions are continued until all strings are terminal strings. The first nonterminal string in Speclang is <comp_spec>. This denotes a complete specification. As mentioned in the previous section a specification developed from Speclang is made up of modules. Each module in itself is a specification. To reflect this the right hand side of the production rule for <comp_spec> is <module_spec>+. In other words, a complete specification is made up of one or more specification modules.

The production rules are as follows:

```
<complete_spec> -> (<module_spec><newline><newline>)

<module_spec> -> <spec_header>
                 (<indent><newline><param_block>)?
                  <indent><newline><spec_block>

<spec_header> -> 'SPEC' <spec_id> 'IS'

<param_block> -> 'PARAMETERS' <indent><newline>
                         <spec_block>
                 'DEFINED_BY' <indent>

<spec_id> -> <identifier>

<spec_block> -> (<extend_form> <indent> <newline>)?
                (<spec_body>|<param_call>)

<spec_body> -> <sort_body> <newline>
               <op_body> <newline>
               <axiom_body>?

<extend_form> -> 'EXTEND' <newline>
                         <indent> <extend_list>
                 'WITH'<indent>

<extend_list> -> ((<spec_id>|<param_call>)';'<newline>)+
<param_call> -> <spec_id> '(' <spec_id> ',' <newline>
                   <indent> 'WHERE' <newline> <indent>
                                <actual_param>

<actual_param> -> <actual_sorts> <newline> <actual_ops>

<actual_sorts> -> 'ACTUAL_SORTS'
         (<newline><indent><sort_id>'IS' <sort_id>';')+

<actual_ops> -> 'ACTUAL_OPS'
           (<newline><indent><op_id>'IS' <op_id>';')+

<sort_body> -> 'SORT' <newline>
                  <indent>(<sort_id>';')+

<sort_id> -> <identifier>

<op_body> -> 'OPERATIONS' <newline>
                 <indent> <primitive_ops> <newline>
                (<indent> <derived_ops> <newline>)?
                (<indent> <error_ops> <newline>)?
                (<indent> <hidden_ops> <newline>)?

<primitive_ops> -> 'PRIMITIVE' <newline>
                   <indent> <operations>

<derived_ops> -> 'DERIVED' <newline> <indent>
                   <operations>

<error_ops> -> 'ERROR' <newline> <indent> <operations>

<hidden_ops> -> 'HIDDEN' <newline> <indent>
                 <operations>

<operations> -> (<op_form> ';' <newline>)+


<op_form> -> <op_id> ':' <sort_list>? '->' <sort_id>

<sort_list> -> (<sort_id> (','<sort_id>)*)?

<axiom_body> -> 'AXIOM' <indent> (<newline> <axioms>)+
   <axioms> -> <expression> '=' <expression> ';'

   <expression> -> (<sort_id>|<op_exp>)

   <op_exp> -> <op_id> ('(' <expression>
                           (','<expression>)* ')')?


------------------------------------------------------------------
Lexical grammar

   <spectext> -> (<delimeter>+)? | (<delimeter_pair>+)?

   <delimeter> -> <comment> | '->' | ';' | '(' | ')' |
                                 ','  | '=' | ':'

   <non_delimeter> -> <identifier> | <string_constant>

   <identifier> -> <letter> (<letter>|<digit>|'_')*

   <string_constant> -> ''' <symbol>* '''

   <letter> -> 'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|
               'K'|'L'|'M'|'N'|'O'|'P'|'Q'|'R'|'S'|'T'|
               'U'|'V'|'W'|'X'|'Y'|'Z'|'a'|'b'|'c'|'d'|
               'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|'n'|
               'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|
               'y'|'z'

   <digit> -> '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

   <newline> -> (<comment>|<carriage_return>)+

   <comment> -> ('!'(<symbol>|<letter>)+ <carriage_return>)+

   <symbol> -> <letter>|<digit>|'|'|'@'|'#'|'$'|'%'|
               '      '|'&'|'*'|'('|')'|'-'|'_'|'+'|
               '!'|'"'|'}'|'|'|'='|'  '|'/'|
               '.'|','|'/'|'{'|'o'|'<'|'>'|';'|
               ':'

   <indent> -> (' '+ | <tab>)
```

LIST OF REFERENCES

1. Tanenbaum, A. S., Klint, P. and Bohm, W., "Guidelines for Software Portability", *Software Practice and Experience*, Vol. 8, 2 March 1978.

2. Goguen, J.A., Thatcher, J.W., and Wagner, E.G.," An Initial Algebra Approach to the Specification of Abstract Data Types, *Current Trends in Programming Methodology*, vcl. 4, Prentice-Hall, 1977.

3. Guttag, J. V., Horrowitz, E. and Musser, D. R., "The Design of Data Type Specifications", *Current Trends in Programming Methodology IV*, Prentice-Hall, 1978.

4. Liskov, B. H. and Zilles, S. N., "Specification Techniques for Data Abstraction", *IEEE Transactions on Software Engineering*, March 1975.

5. Fasel, J. H., *Programming Languages as Abstract Data Types*, Ph.D. dissertation, Purdue University, Lafayette, Indiana, August 1980.

6. Majster, M. E., "Limits of the Algebraic Specification of Data Types", *Sigplan Notice 12*, 10 Oct 1977.

7. Linden, T. A., "Specifying Abstract Data Types by Restriction", *Software Engineering Notes 3*, 2 April 1978.

8. Burstall, R. M., Goguen, J. A., *Putting Theories Together to Make Specifications*, International Joint Conference on Artificial Intelligence, 5th, MIT 1977.

9. Ganzinger, H., "Parameterized Specifications: Parameter Passing and Implementation with Respect to Observability", *ACM Transactions on Programming Languages and Systems*, Vol 5. No. 3, July 1983.

10. Davis, D. L., *Syntax Directed Editing*, Unpublished notes on syntax directed editors, June 1984.

11. MacLennan, B. J., *The Automatic Generation of Syntax Directed Editors*, Technical Notes, October 1981.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
Cmaeron Station
Alexandria, Virginia 22314

2. Library Code 0142 2
Naval Postgraduate School
Monterey, California 93943

3. Department Chairman, Code 52 2
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943

4. Curricular Officer, Code 37 1
Computer Technology Curricular Office
Naval Postgraduate School
Monterey, California 93943

5. Associate Professor Daniel L. Davis, Code 52Vv 2
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943

6. Professor Gordon H. Bradley, Code 52Bz 1
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943

7. Lcdr Norvell L. Lilly 2
VAW 112
FPO San Francisco, California 96601

8

210361

Thesis
L63852 Lilly
c.1 An algebraic specification language and a syntax directed editor.